Universidad Autónoma de Nuevo León

Graduate Program in Systems Engineering

Experimental Portfolio

Network Flows Optimization

January-June, 2019

Oscar Alejandro Hernández López

Homework 1 Corrections

In this homework corrections in notations and references were made, as well as in bibliography, and captions of figures added after the feedback.

Homework Assignment 1: Network Flows

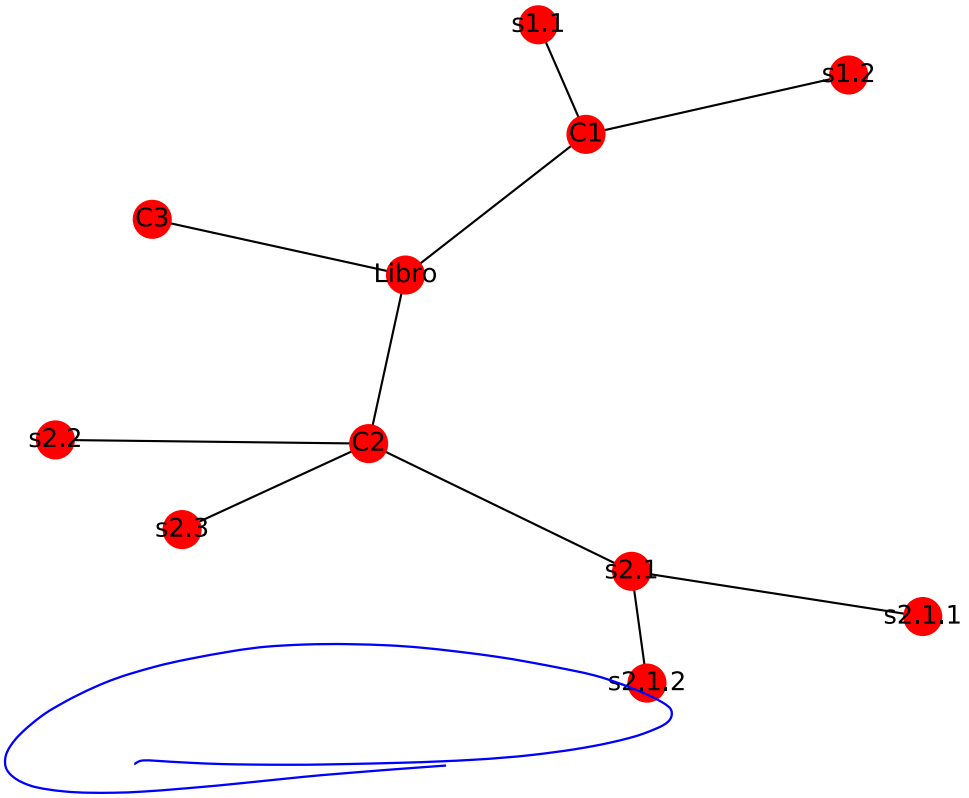
Oscar Alejandro Hernández López

Undirected acyclic graph

Consider the general index of a book. Such index is a tree and has a relation with the chapters and these with the sections. The root, the node *Book*, has three subtrees which are the chapters *C1*,*C2* and *C3*. This three nodes are the inheritance of the node named *Book*. Also we have two subtrees with non trivial relations, for example *C2* has another three subtrees which represent sections *s2.1*, *s2.2*, *s2.3*. [1]

```python
import networkx as nx                                   #Library to create graphs
import matplotlib.pyplot as plt                         #Library to show graphs

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')])   #Add a list of edges
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')])

nx.draw(G,with_labels=True)                             #Draw the Graph named G
plt.savefig("Graph01.eps", format="EPS")                #Save figure in eps format
plt.show()                                              #Show (Print) Graph
```

01undirected_acyclic_graph.py

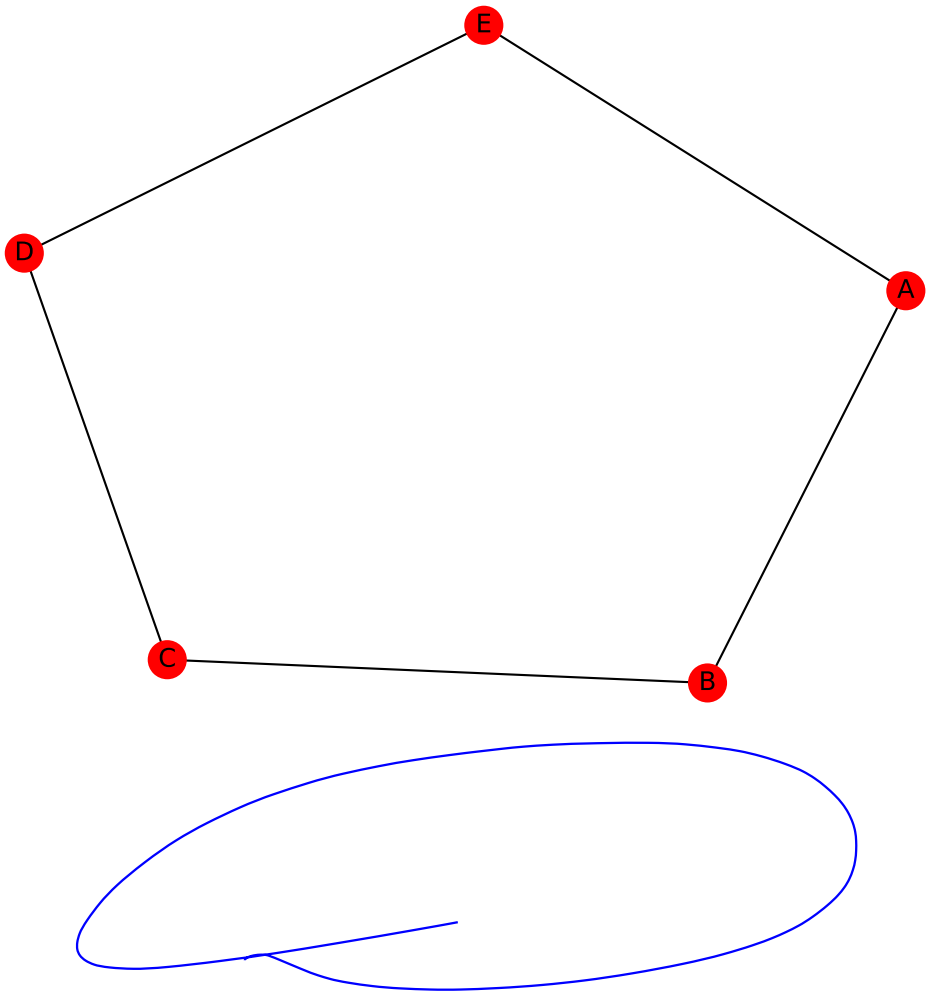


Undirected cyclic graph

In this situation we have five cities. Cities A and B are separated by a distance of 50 km, between cities B and C are 20 km and there is 30 km and 35 km between cities C and D, and D and E, respectively. The nearest distance is between cities A and E, with 15 km. [6]

```
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, with_labels=True)
plt.savefig("Graph02.eps", format="EPS")
plt.show()
```

02undirected_cyclic_graph.py

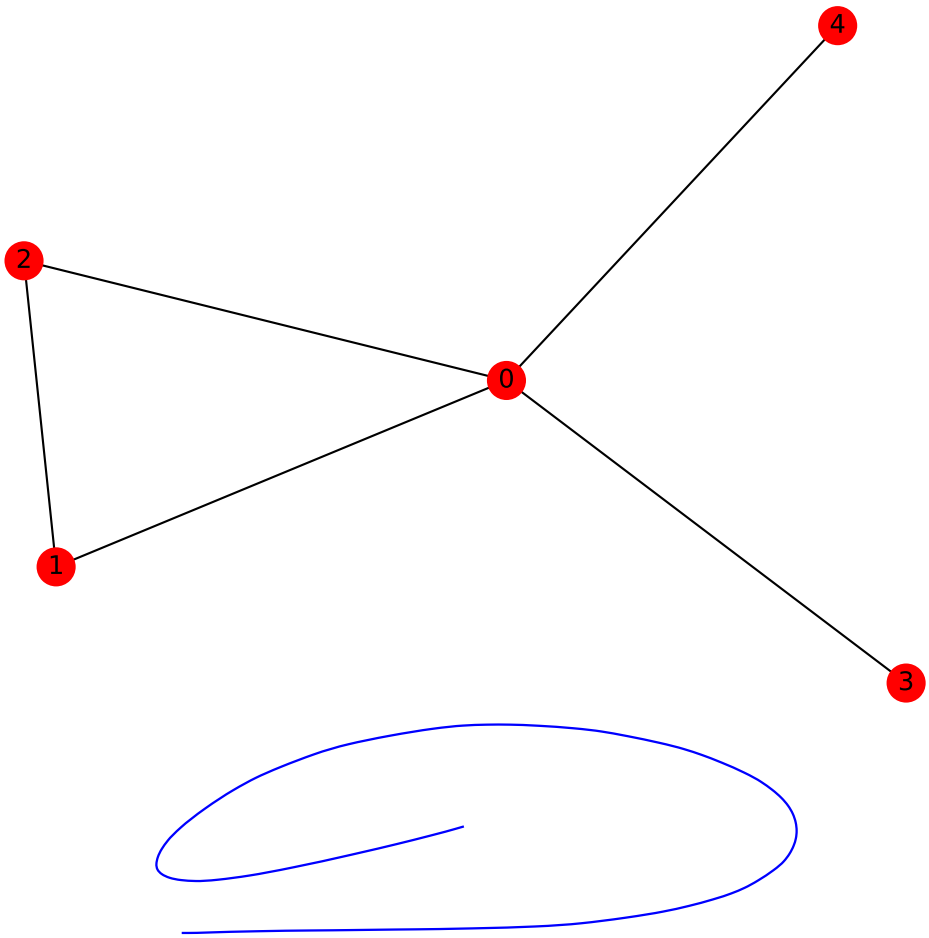


Undirected reflexive graph

The example shows relationships between republics, we can call it cluster (*0,1* and *2*). Clusters *3* and *4* are other republics that have significant ties with *0*, but almost no ties with anyone else. In case of republic *3* in addition to other relationships, can do other local relations or trades. [2]

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

H=nx.Graph()

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

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

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

03undirected_reflexive_graph.py

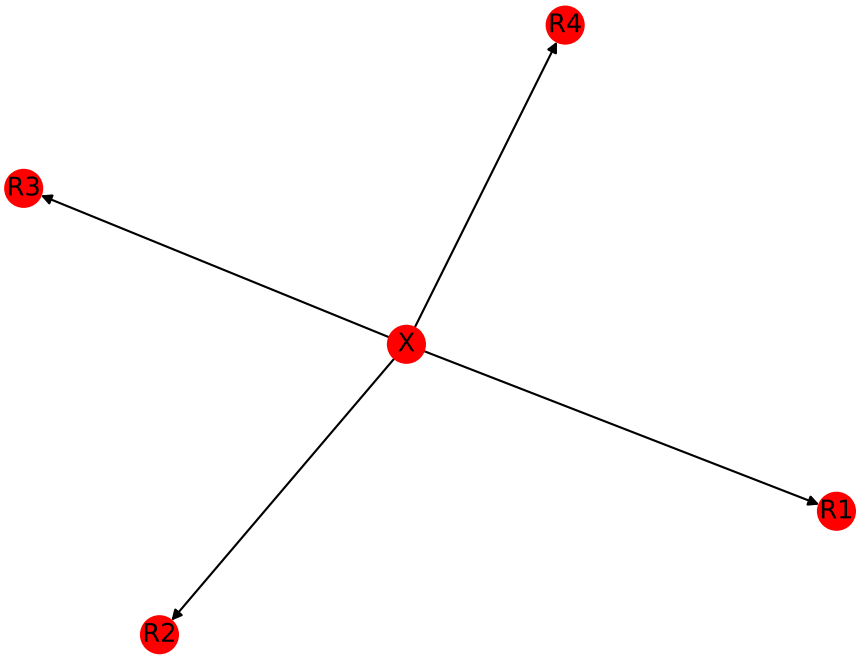


Directed acyclic graph

The example is about the Bayes's basic structure, which begins by a tree structure with its predictive variables and later connect the X variable with every single predictive variables R_n. In this acyclic directed graph every node represents a variable and every edge a probabilistic dependence, specifying the conditional probability of this variables with its sources. Every connections are identified by simple path that does not repeat vertices. [9]

```
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")])
nx.draw(G,with_labels=True)
plt.savefig("Graph04.eps", format="EPS")
plt.show()
```

04directed_acyclic_graph.py

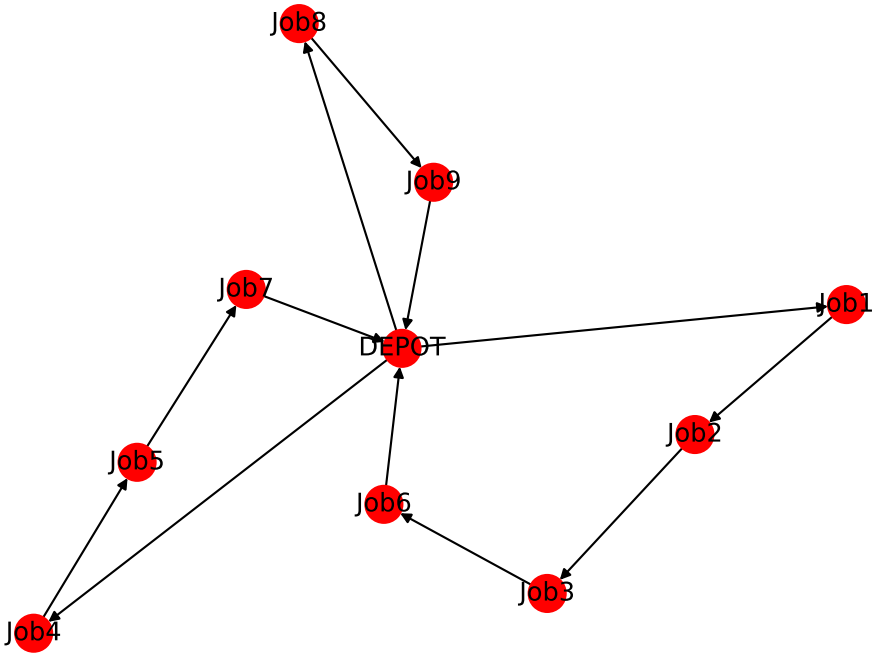


Directed cyclic graph

This situation consists in a fleet of homogeneous vehicles which need to distribute the jobs to respective customers. The routing graph is the set of nodes where a node *DEPOT* represents the main facility where the vehicles are loaded. The rest of nodes denote the rest of the customers. Each job corresponds to a unique customer. The number of vehicles is limited and each customer order needs to be delivered. Tours are allowed to visit multiple customers. [8]

```python
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'])

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

05directed_ciclic_graph.py

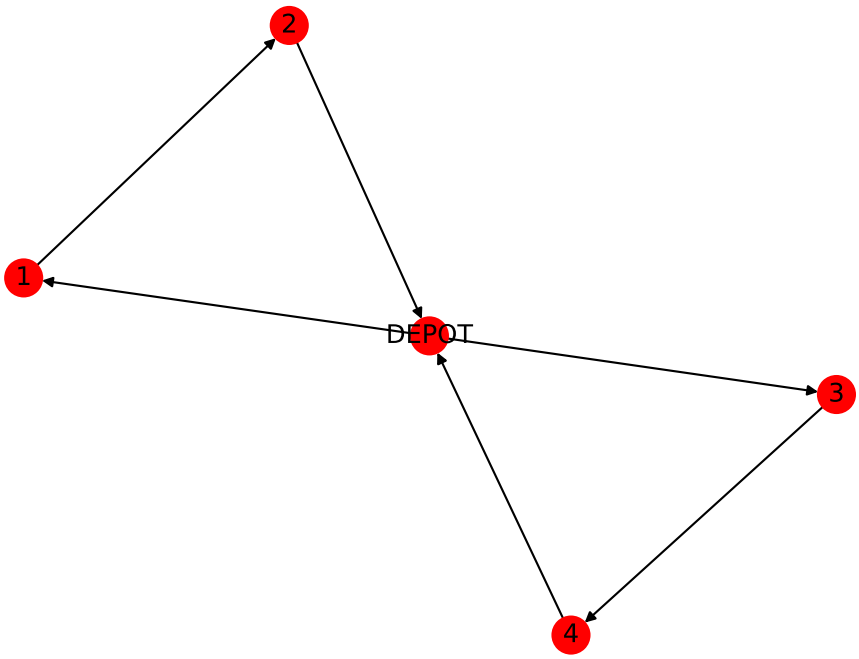


Directed reflexive graph

A global courier has to deliver some goods in 4 countries. For technical issues and distances between destinations a transport can only visit at most two destinations. Once it visit the costumers has to return to depot in order to begin maintenance for the next departure. In case an order had started from *DEPOT* and is cancelled the transport can return. [7]

```
1  import networkx as nx
2  import matplotlib.pyplot as plt
3
4  R=nx.DiGraph()
5  R.add_nodes_from(["DEPOT","1","2","3","4"])
6
7  R.add_edges_from([("DEPOT","1"),("1","2"),("2","DEPOT")])
8  R.add_edges_from([("DEPOT","3"),("3","4"),("4","DEPOT"),("DEPOT","DEPOT")])
9
10 nx.draw(R, with_labels=True)
11 plt.savefig("Graph06.eps", format="EPS")
12 plt.show()
```

06directed_reflexive_graph.py

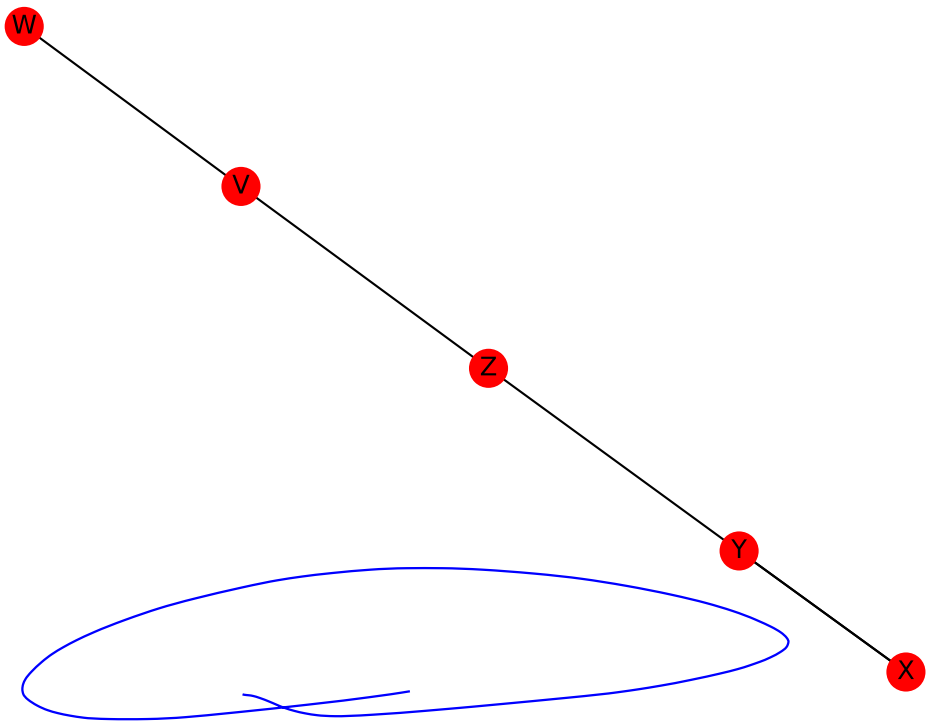


Undirected acyclic multigraph

Here we have a set of 5 tasks: X, Y, Z, V, W which need to be done to run a system. We also can consider we have 6 processors we need to assign to each task. Due to complexity of activities X and Y we shall assign two processors to execute at least one of them. This way we have two different relations between X and Y, and only one relation between the other tasks. [5]

```
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_edges_from([('X','Y'),('X','Y'),('Y','Z'),('Z','V'),('V','W')])

nx.draw(M, with_labels=True)
plt.savefig("Graph07.eps", format="EPS")
plt.show()
```

07undirected_acyclic_multigraph.py

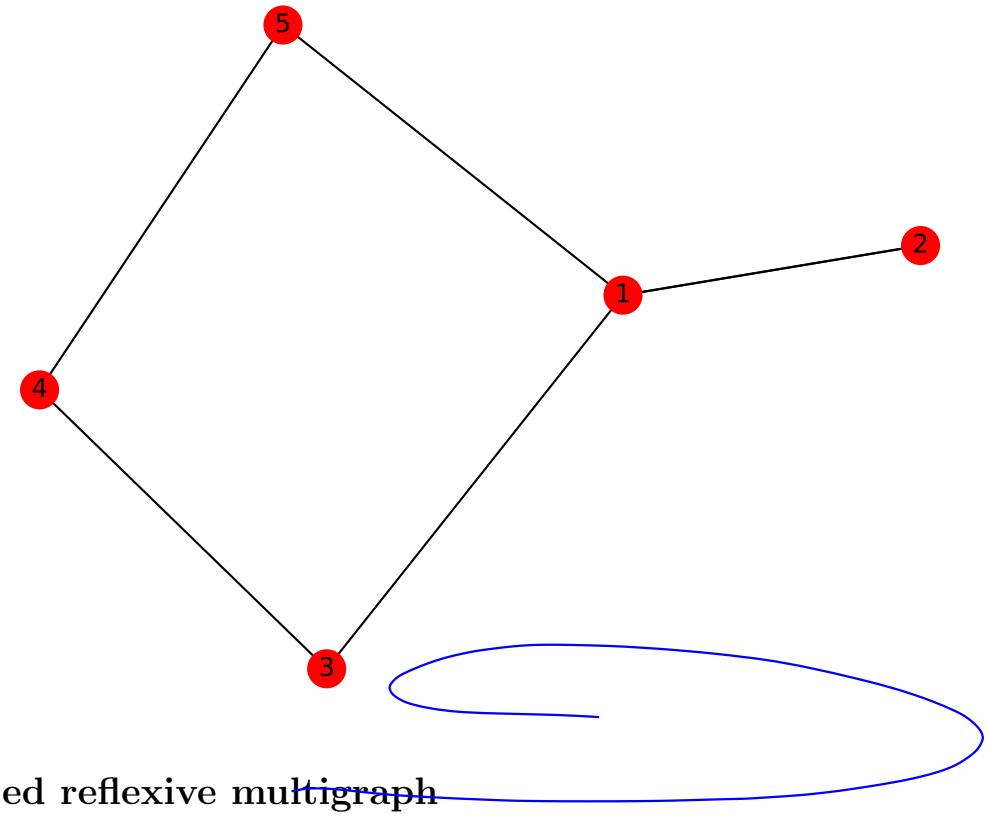


Undirected cyclic multigraph

This example is about a city divided in five zones. This five zones are united by six roads and people of the city, wants to cross each of the paths only once and end in the same place where they have started. [11]

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

M=nx.MultiGraph()

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

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

nx.draw(M, with_labels=True)
plt.savefig("Graph08.eps", format="EPS")
plt.show()
```

08undirected_cyclic_multigraph.py

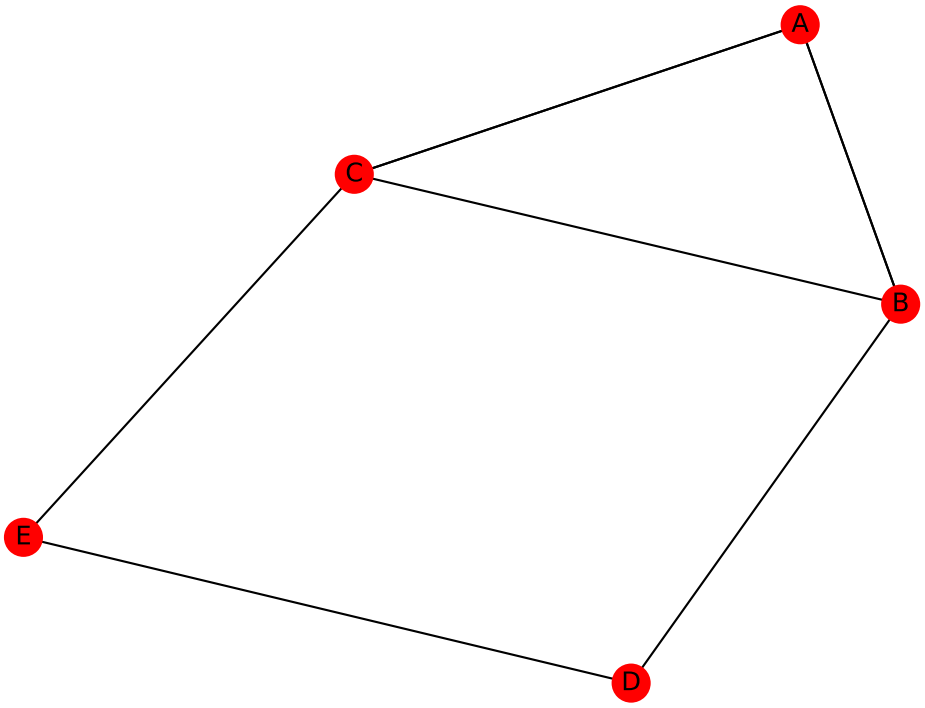


Undirected reflexive multigraph

Assume we have five workstations in a plant. The product needs to go through all the stations not matter the order or a secuence. We know station A has a section of quality control where the product is tested, and if it is not good enough needs to be reprocessed. Also relations in sections A-B or A-C can be choosing two paths, while relations B-C, B-D,C-E and D-E can only be performed by one path. [12]

```
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","A"),("A","B"),("A","B"),("A","C"),("A","C"),("B","C"),("B","D"),("C","E"),("D","E")])

nx.draw(M, with_labels=True)
plt.savefig("Graph09.eps", format="EPS")
plt.show()
```

09undirected_reflexive_multigraph.py

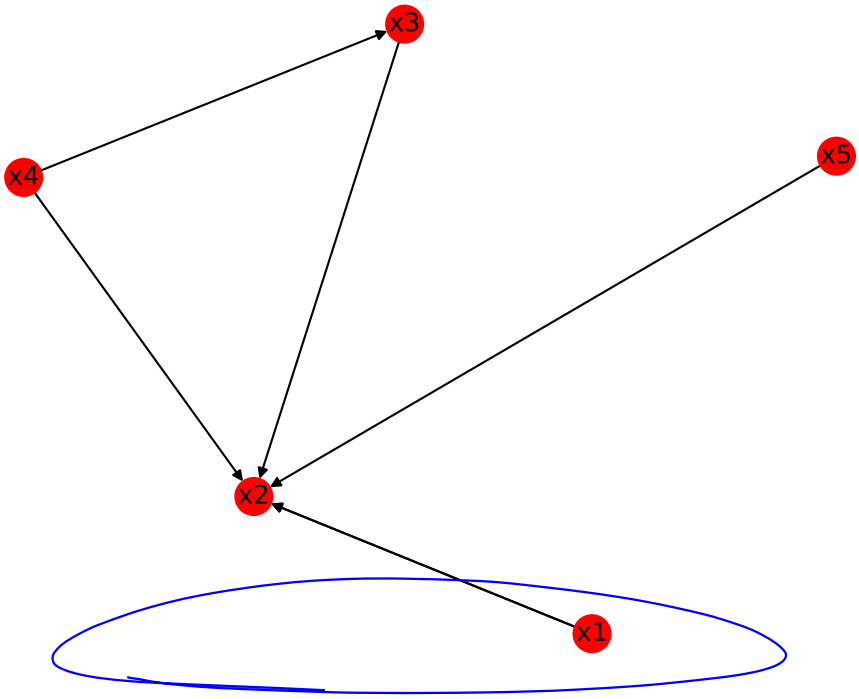


Directed acyclic multigraph

Certain company has 5 departments which some of them share relations in terms of information flow. Let say x_2 is the central deparment and needs to have a complete feedback. Deparments x_1 and x_5 has to share information only with the central deparment, the same as x_3, with the difference of two subjects. Whereas the department x_4 has to share information to departments x_3 and the central one. [3]

```
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_edges_from([("x1","x2"),("x1","x2"),("x3","x2"),("x4","x3"),("x4","x2"),("x5","x2")])

nx.draw(M, with_labels=True)
plt.savefig("Graph10.eps", format="EPS")
plt.show()
```

10directed_acyclic_multigraph.py

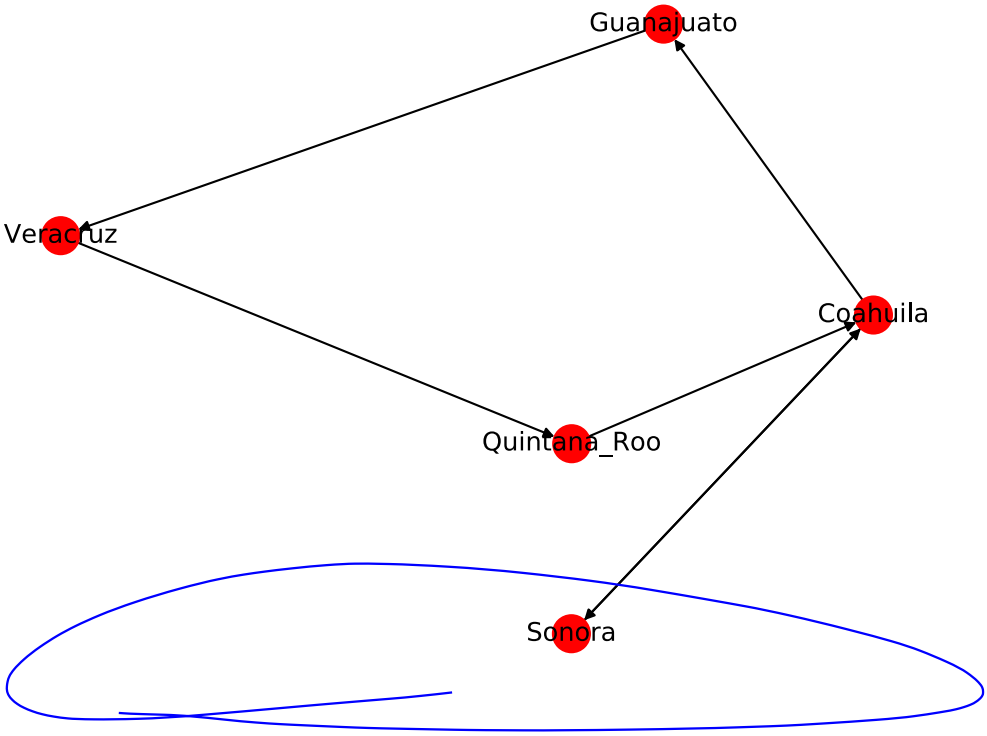


Directed cyclic multigraph

A student wants to spend his summer vacations in 5 states of Mexico: Coahuila, Sonora, Guanajuato, Veracruz and Quintana Roo. He will start in Coahuila, and then he will go to Sonora to visit his family. Then he will return to Coahuila to travel with his friends to Guanajuato, Veracruz and Quintana Roo in that order. He has the chance to return to Coahuila once he finish the visits. [4]

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

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

M.add_edges_from([("Coahuila","Sonora"),("Sonora","Coahuila"),("Coahuila","Guanajuato"),("
    Guanajuato","Veracruz"),("Veracruz","Quintana_Roo"),("Quintana_Roo","Coahuila")])

nx.draw(M,with_labels=True)
plt.savefig("Graph11.eps", format="EPS")
plt.show()
```

11directed_cyclic_multigraph.py

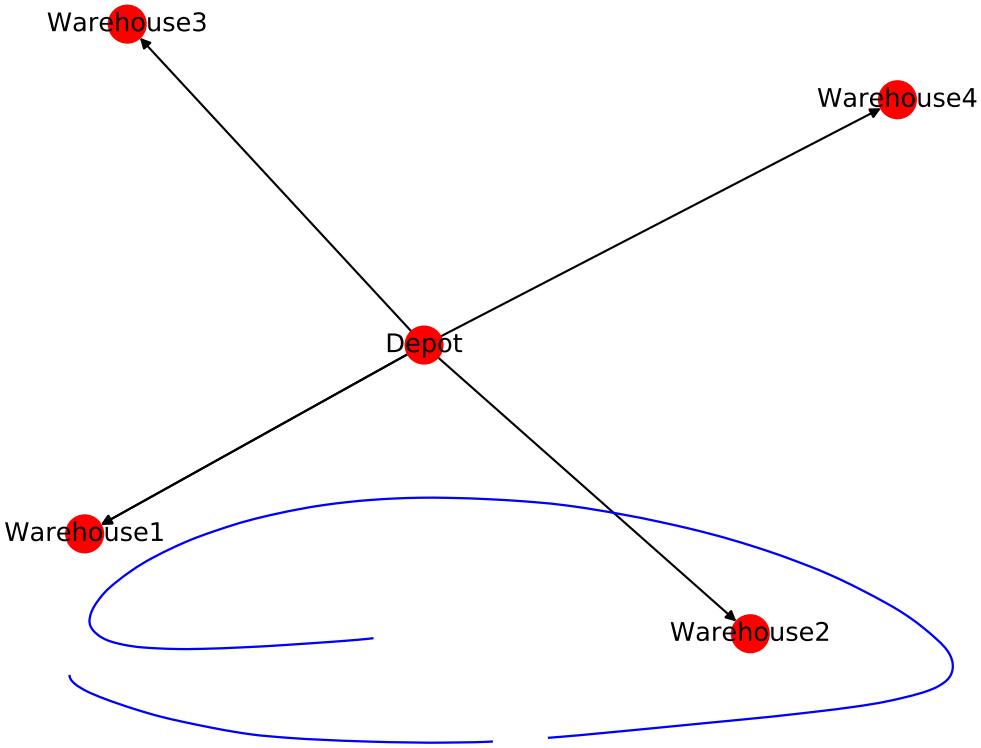


Directed reflexive multigraph

A logistic company has to deliver 4 orders of pallets from the depot to its warehouses. In order to speed operations up it will allocate 2 trucks to deliver in *warehouse1* and will use only one truck in the other warehouses. In order to keep stock, it will have to produce in-place a small quantity of pallets for inventory security and transport it to the small warehouse at the depot.[10]

```
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")])

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

12directed_reflexive_multigraph.py



References

[1] Ullman J.E. Aho A.V., Hopcroft J.E. *Estructura de Datos y Algoritmos*. Addison-Wesley, 1988.

[2] Maksim Tsvetovat Alexander Kouznetsov. Social network analysis for startups, 2019.

[3] Romel Félix Capcha Ventura. La teoría de grafos en la resolución de problemas aritméticos para estudiantes del laboratorio de investigación e innovación pedagógica de la universidad nacional daniel alcides carrión de pasco-2014. 2016.

[4] Norma Cabrera Luisa Gonzalez Carlos Garcia, Abel Rodriguez. Detection and correction of inconsistencies of cyclical references in database logical schemas. *Fac. Ing. Univ. Antioquia*, (55):165–173, 2010.

[5] Edgardo Ferro, Javier D Orozco, and Ricardo Cayssials. Asignación de tareas en un sistema distribuido de tiempo real duro. In *II Congreso Argentino de Ciencias de la Computación*, 1996.

[6] Zamantha Gonzalez. Grafos. pages 22–25, 2008.

[7] Jim Webber Ian Robinson and Emil Eifrem. *Graph Databases*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2 edition, 2015.

[8] Saadettin Erhan Kesen and Tolga Bektaş. Integrated production scheduling and distribution planning with time windows. In *Lean and Green Supply Chain Management*, pages 231–252. Springer, 2019.

[9] Carlos Arturo Vega Lebrun. Integracion de herramientas de tecnologias de informacion como soporte en la administracion del conocimiento, 2005.

[10] Jesus Garcia Miranda. *Matematica Discreta*. 2005.

[11] Sherman K. Stein. The mathematician as an explorer. *Scientific American*, 204(5):148–161, 1961.

[12] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

Homework Assignment 1: Network Flows

5273

Undirected acyclic graph

Consider the general index of a book. Such index is a tree and has a relation with the chapters and these with the sections. The root, the node b, has three subtrees which are the chapters c_1,c_2 and c_3. This three nodes are the inheritance of the node named b. Also we have two subtrees with non trivial relations, for example c_2 has another three subtrees which represent sections $s_{2.1}$, $s_{2.2}$, $s_{2.3}$ [1].

```python
import networkx as nx                                   #Library to create graphs
import matplotlib.pyplot as plt                         #Library to show graphs

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

G.add_node('b')                                     #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([('b','c1',),('b','c2',),('b','c3')])   #Add a list of edges
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')])

nx.draw(G,with_labels=True)                             #Draw the Graph named G
plt.savefig("Graph01.eps", format="EPS")                #Save figure in eps format
plt.show()          #Show (Print) Graph
```

01undirected_acyclic_graph.py

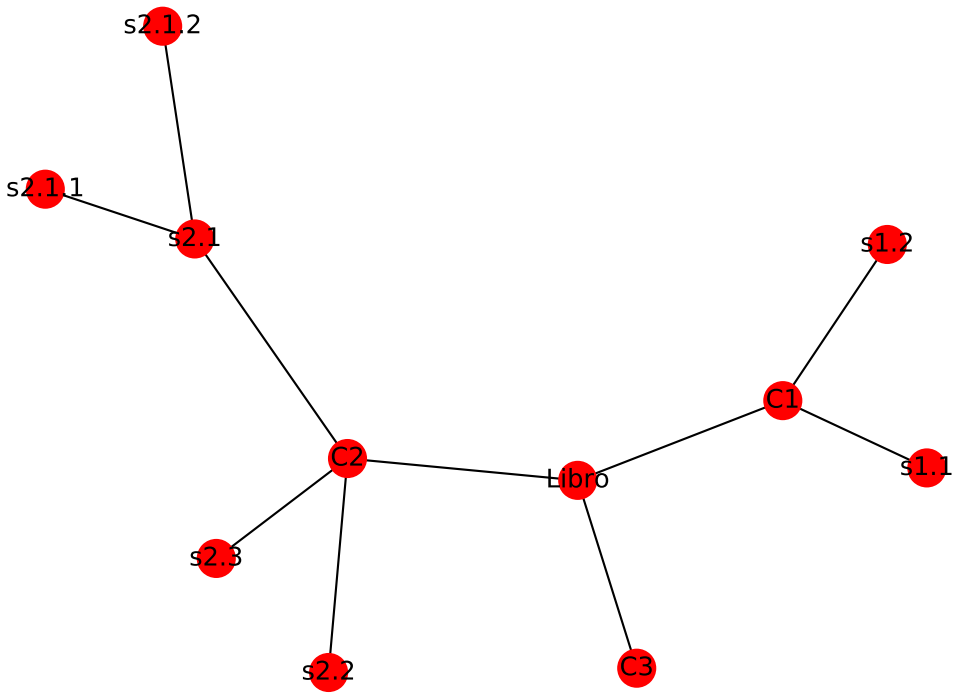


Figure 1: Undirected acyclic graph

Undirected cyclic graph

In this situation we have five cities. Cities A and B are separated by a distance of 50 km, between cities B and C are 20 km and there is 30 km and 35 km between cities C and D, and D and E, respectively. The nearest distance is between cities A and E, with 15 km. [6]

```
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, with_labels=True)
plt.savefig("Graph02.eps", format="EPS")
plt.show()
```

02undirected_cyclic_graph.py

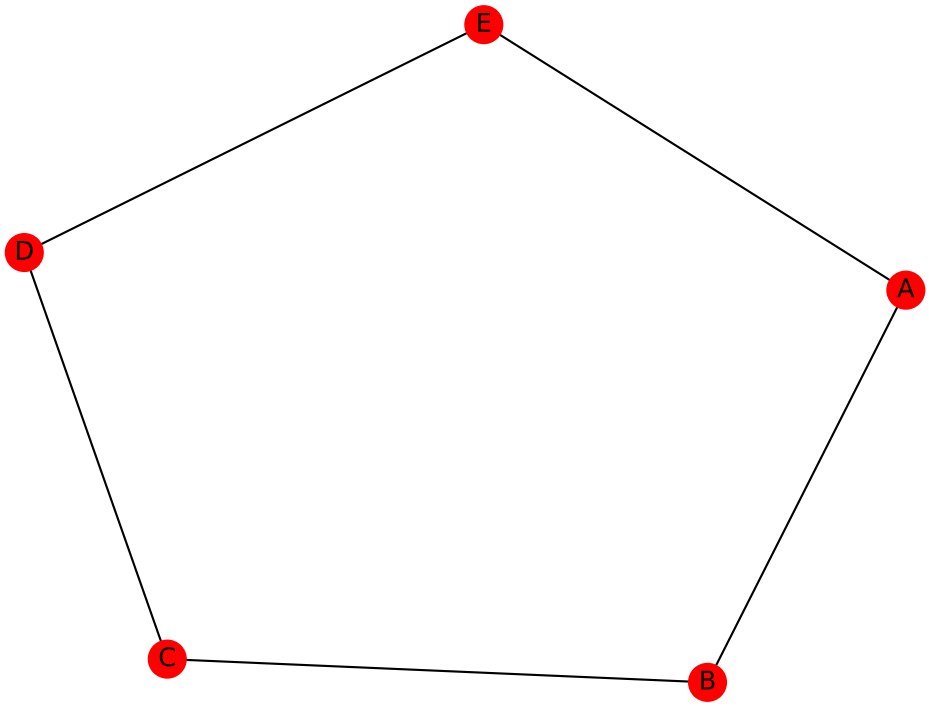


Figure 2: Undirected cyclic graph

Undirected reflexive graph

The example shows relationships between republics, we can call it cluster (0, 1 and 2). Clusters 3 and 4 are other republics that have significant ties with 0, but almost no ties with anyone else. In case of republic 3 in addition to other relationships, can do other local relations or trades [2].

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

H=nx.Graph()

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

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

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

03undirected_reflexive_graph.py

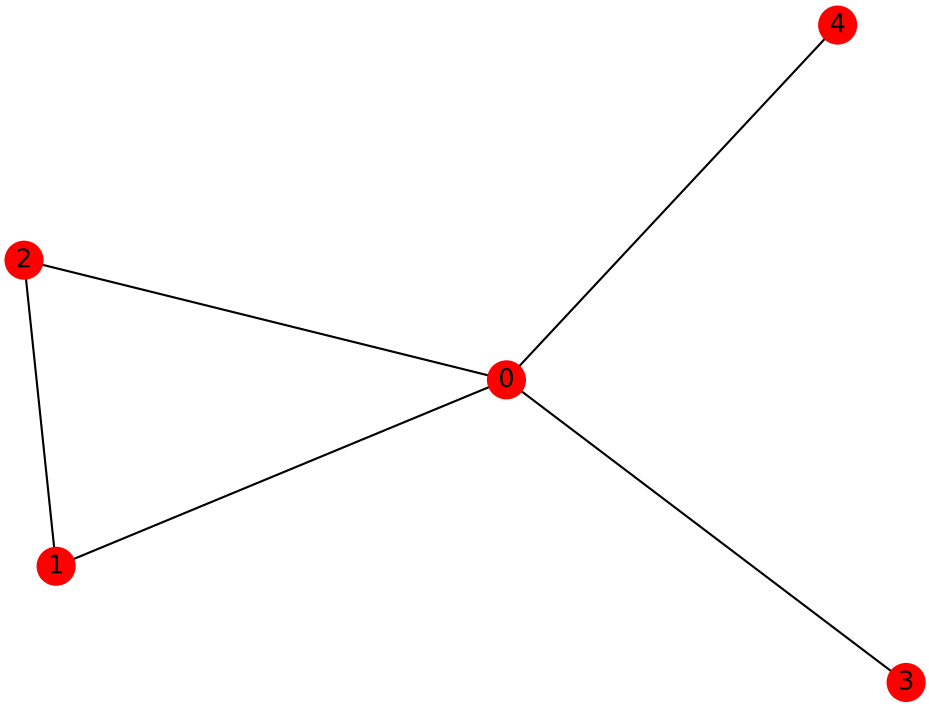


Figure 3: Undirected reflexive graph

Directed acyclic graph

The example is about the Bayes's basic structure, which begins by a tree structure with its predictive variables and later connect the x variable with every single predictive variables r_n. In this acyclic directed graph every node represents a variable and every edge a probabilistic dependence, specifying the conditional probability of this variables with its sources. Every connections are identified by simple path that does not repeat vertices [9].

```
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")])
nx.draw(G, with_labels=True)
plt.savefig("Graph04.eps", format="EPS")
plt.show()
```

04directed_acyclic_graph.py

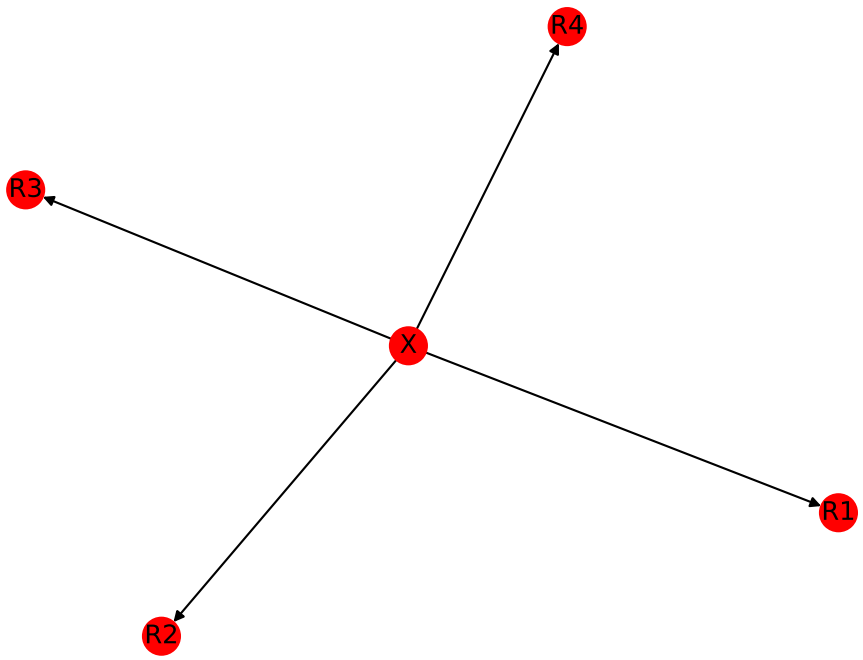


Figure 4: Directed acyclic graph

Directed cyclic graph

This situation consists in a fleet of homogeneous vehicles which need to distribute the jobs to respective customers. The routing graph is the set of nodes where a node d represents the main facility where the vehicles are loaded (depot). The rest of nodes denote the rest of the customers. Each job corresponds to a unique customer. The number of vehicles is limited and each customer order needs to be delivered. Tours are allowed to visit multiple customers [8].

```
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'])

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

05directed_ciclic_graph.py

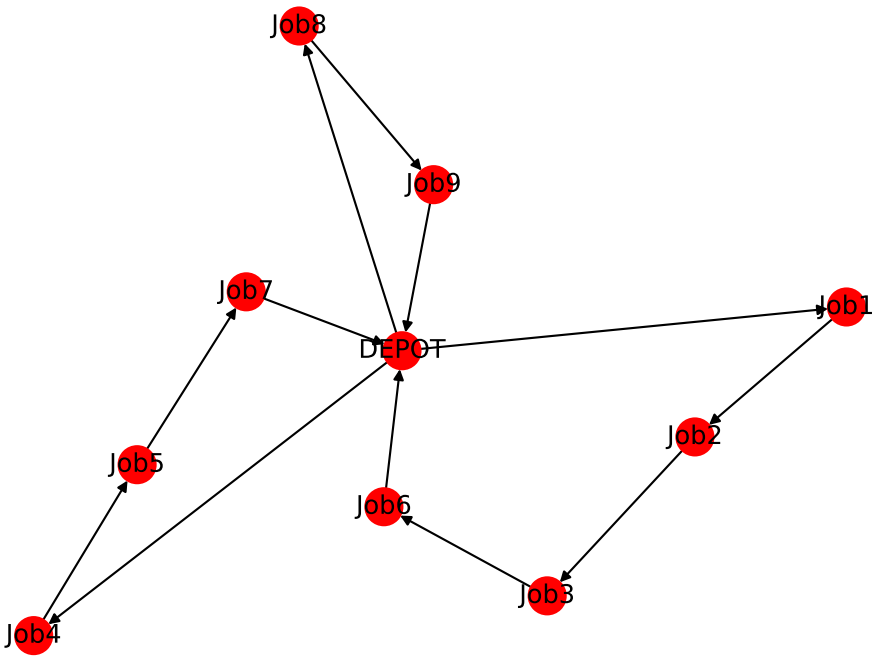


Figure 5: Directed cyclic graph

Directed reflexive graph

A global courier has to deliver some goods in 4 countries. For technical issues and distances between destinations a transport can only visit at most two destinations. Once it visits the costumers has to return to depot in order to begin maintenance for the next departure. In case an order had started from d and is cancelled the transport can return [7].

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

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

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

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

06directed_reflexive_graph.py

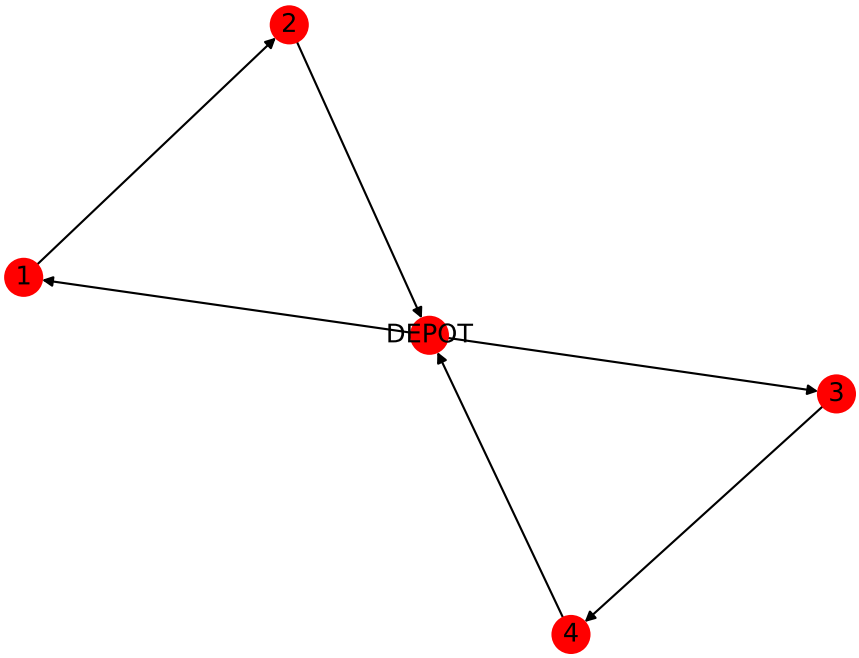


Figure 6: Directed reflexive graph

Undirected acyclic multigraph

Here we have a set of five tasks: x, y, z, v, w which need to be done to run a system. We also can consider we have six processors we need to assign to each task. Due to complexity of activities x and y we shall assign two processors to execute at least one of them. This way we have two different relations between x and y, and only one relation between the other tasks [5].

```
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_edges_from([('X','Y'),('X','Y'),('Y','Z'),('Z','V'),('V','W')])

nx.draw(M, with_labels=True)
plt.savefig("Graph07.eps", format="EPS")
plt.show()
```

07undirected_acyclic_multigraph.py

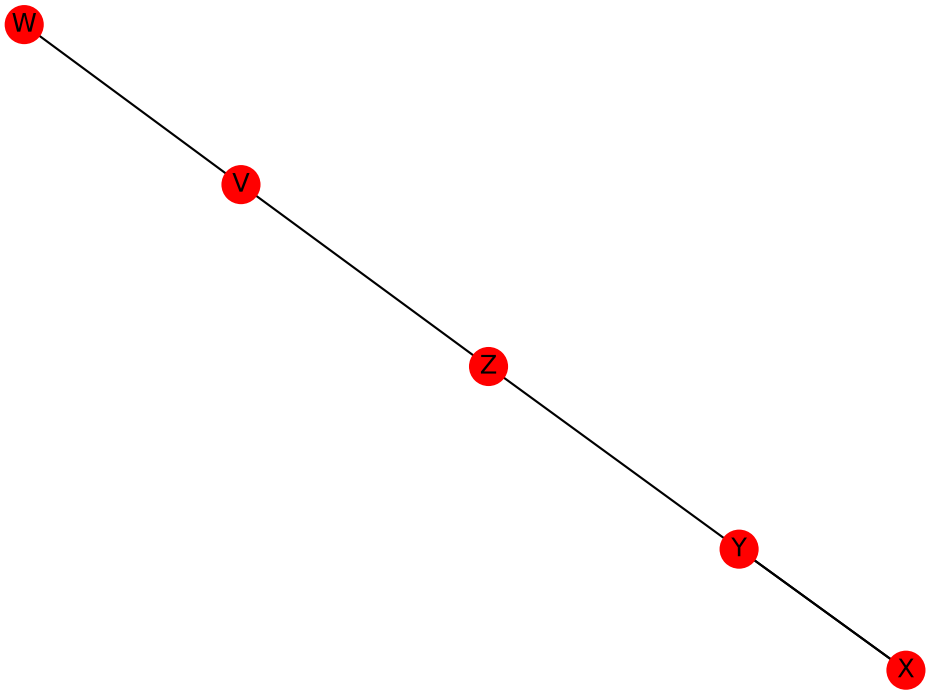


Figure 7: Undirected acyclic multigraph

Undirected cyclic multigraph

This example is about a city divided in five zones. This five zones are united by six roads and people of the city, wants to cross each of the paths only once and end in the same place where they have started [11].

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

M=nx.MultiGraph()

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

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

nx.draw(M, with_labels=True)
plt.savefig("Graph08.eps", format="EPS")
plt.show()
```

08undirected_cyclic_multigraph.py

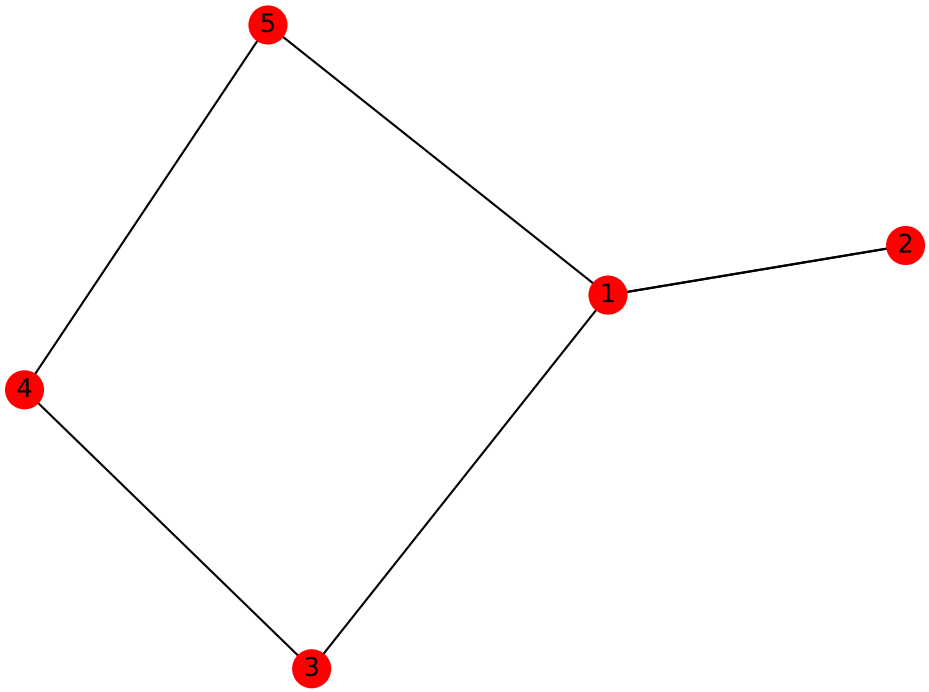


Figure 8: Undirected cyclic multigraph

Undirected reflexive multigraph

Assume we have five workstations in a plant. The product needs to go through all the stations not matter the order or a secuence. We know station a has a section of quality control where the product is tested, and if it is not good enough needs to be reprocessed. Also relations in sections a-b or a-c can be choosing two paths, while relations b-c, b-d, c-e and d-e can only be performed by one path [12].

```
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","A"),("A","B"),("A","B"),("A","C"),("A","C"),("B","C"),("B","D"),("C","E"),("D","E")])

nx.draw(M, with_labels=True)
plt.savefig("Graph09.eps", format="EPS")
plt.show()
```

09undirected_reflexive_multigraph.py

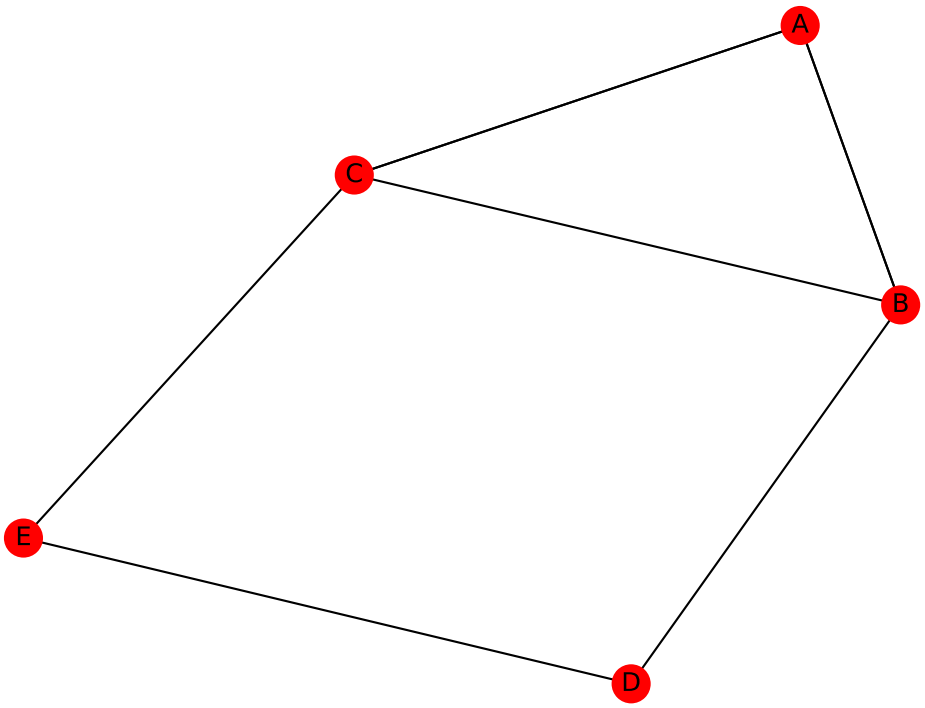


Figure 9: Undirected reflexive multigraph

Directed acyclic multigraph

Certain company has five departments which some of them share relations in terms of information flow. Let say x_2 is the central deparment and needs to have a complete feedback. Deparments x_1 and x_5 has to share information only with the central deparment, the same as x_3, with the difference of two subjects, whereas the department x_4 has to share information to departments x_3 and the central one [3].

```
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_edges_from([("x1","x2"),("x1","x2"),("x3","x2"),("x4","x3"),("x4","x2"),("x5","x2")])

nx.draw(M, with_labels=True)
plt.savefig("Graph10.eps", format="EPS")
plt.show()
```

10directed_acyclic_multigraph.py

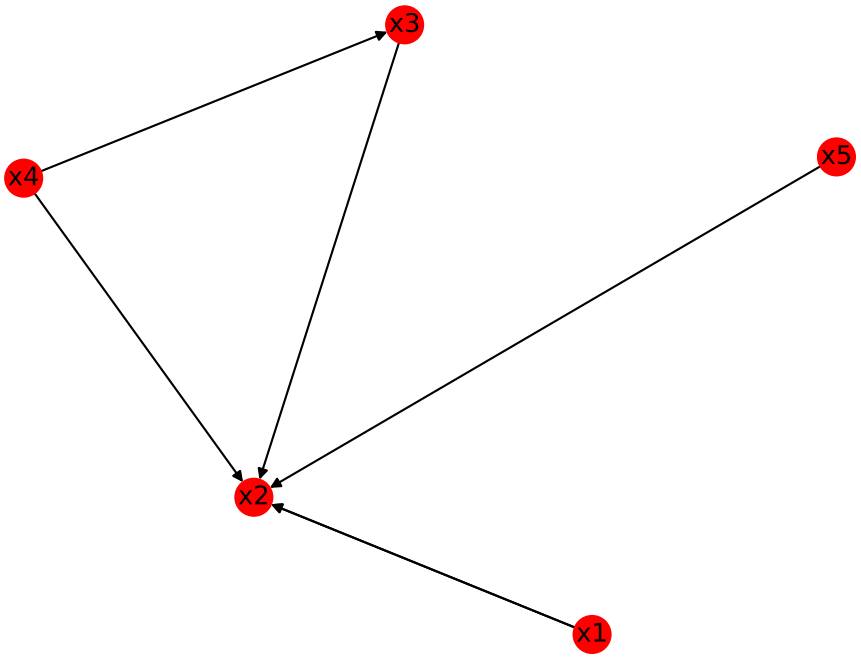


Figure 10: Directed acyclic multigraph

Directed cyclic multigraph

A student wants to spend his summer vacations in five states of Mexico: Coahuila, Sonora, Guanajuato, Veracruz and Quintana Roo. He will start in Coahuila, and then he will go to Sonora to visit his family. Then he will return to Coahuila to travel with his friends to Guanajuato, Veracruz and Quintana Roo in that order. He has the chance to return to Coahuila once he finish the visits [4].

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

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

M.add_edges_from([("Coahuila","Sonora"),("Sonora","Coahuila"),("Coahuila","Guanajuato"),("
    Guanajuato","Veracruz"),("Veracruz","Quintana_Roo"),("Quintana_Roo","Coahuila")])

nx.draw(M, with_labels=True)
plt.savefig("Graph11.eps", format="EPS")
plt.show()
```

11directed_cyclic_multigraph.py

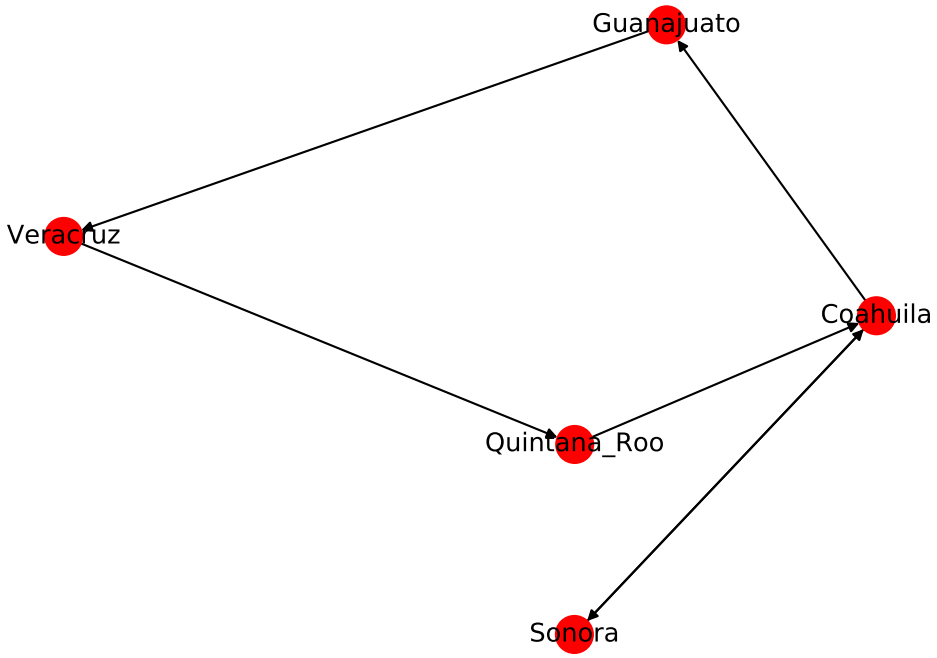


Figure 11: Directed cyclic multigraph

Directed reflexive multigraph

A logistic company has to deliver four orders of pallets from the depot to its warehouses. In order to speed operations up it will allocate two trucks to deliver in *warehouse1* and will use only one truck in the other warehouses. In order to keep stock, it will have to produce in-place a small quantity of pallets for inventory security and transport it to the small warehouse at the depot [10].

```
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")])

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

12directed_reflexive_multigraph.py

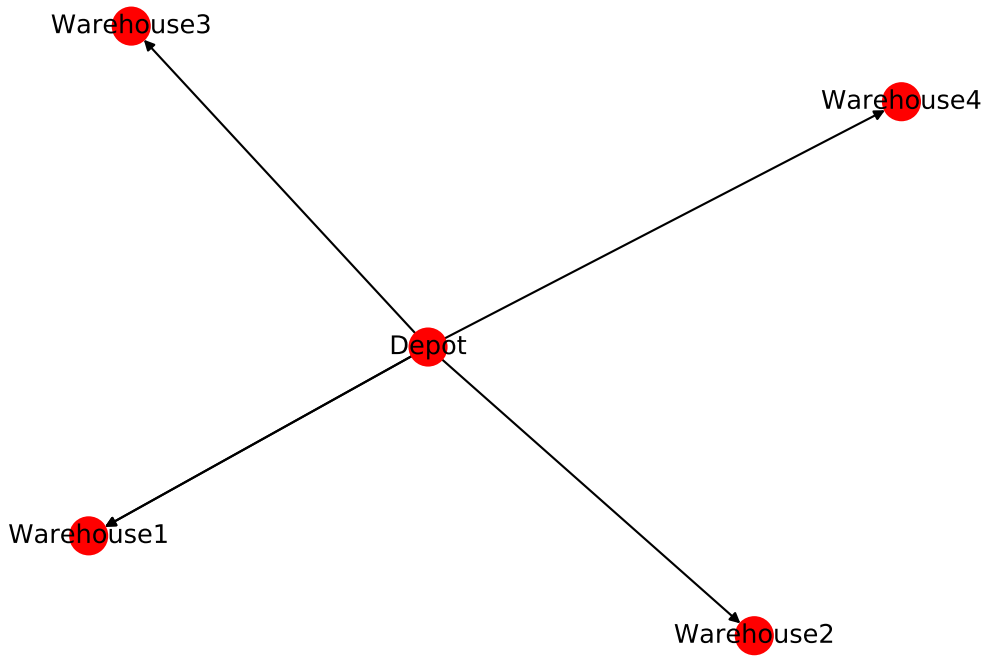


Figure 12: Directed reflexive multigraph

References

[1] Ullman J.E. Aho A.V., Hopcroft J.E. *Estructura de Datos y Algoritmos*. Addison-Wesley, 1988.

[2] Maksim Tsvetovat Alexander Kouznetsov. Social network analysis for startups, 2019. `https://www.oreilly.com/library/view/social-network-analysis/9781449311377/ch04.html`, Last accessed on 2019-02-10.

[3] Romel Félix Capcha Ventura. La teoría de grafos en la resolución de problemas aritméticos para estudiantes del laboratorio de investigación e innovación pedagógica de la Universidad Nacional Daniel Alcides Carrión de Pasco. 2016.

[4] Norma Cabrera, Luisa González, Carlos García, Abel Rodríguez. Detection and correction of inconsistencies of cyclical references in database logical schemas. *Facultad de Ingeniería. Universidad de Antioquia*, (55): 165–173, 2010.

[5] Edgardo Ferro, Javier D Orozco, and Ricardo Cayssials. Asignación de tareas en un sistema distribuido de tiempo real duro. In *II Congreso Argentino de Ciencias de la Computación*, 1996.

[6] Zamantha Gonzalez. Grafos. pages 22–25, 2008.

[7] Jim Webber Ian Robinson and Emil Eifrem. *Graph Databases*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2 edition, 2015.

[8] Saadettin Erhan Kesen and Tolga Bektaş. Integrated production scheduling and distribution planning with time windows. In *Lean and Green Supply Chain Management*, pages 231–252. Springer, 2019.

[9] Carlos Arturo Vega Lebrun. Integración de herramientas de tecnologías de información como soporte en la administración del conocimiento, 2005.

[10] Jesus Garcia Miranda. *Matematica Discreta. Introduccion a la Teoria de Grafos*, pages 131–133, 2005.

[11] Sherman K. Stein. The mathematician as an explorer. *Scientific American*, 204(5):148–161, 1961.

[12] Douglas Brent West. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

Homework 2 Corrections

In this homework corrections in references were made. Errors in misused words were corrected. Another notations were also fixed as well as other mistake in the bibliography.

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].

```python
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

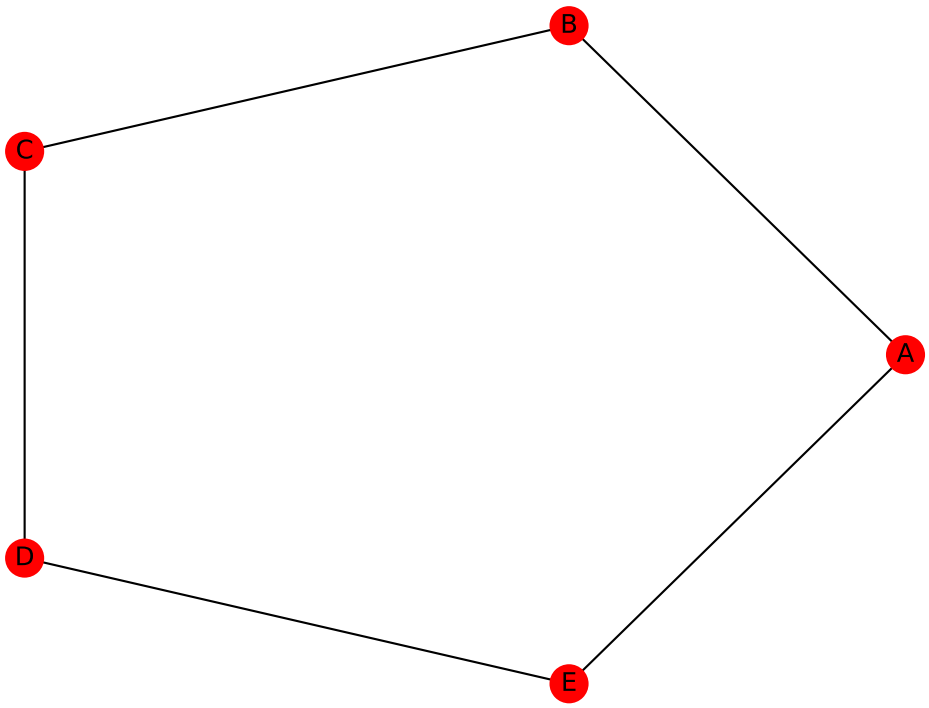


Figure 1: Circular layout

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].

```
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

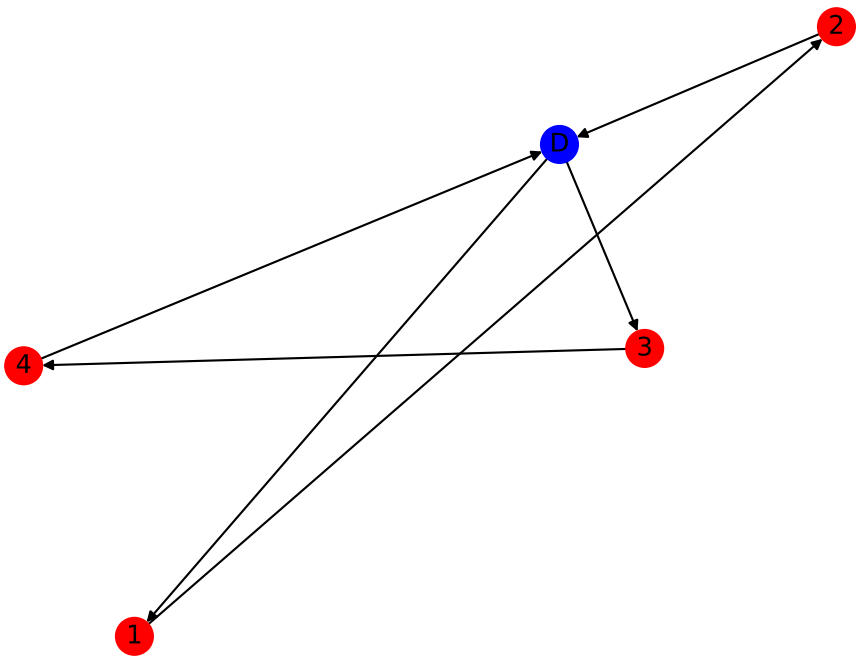


Figure 2: Random layout

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].

```
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

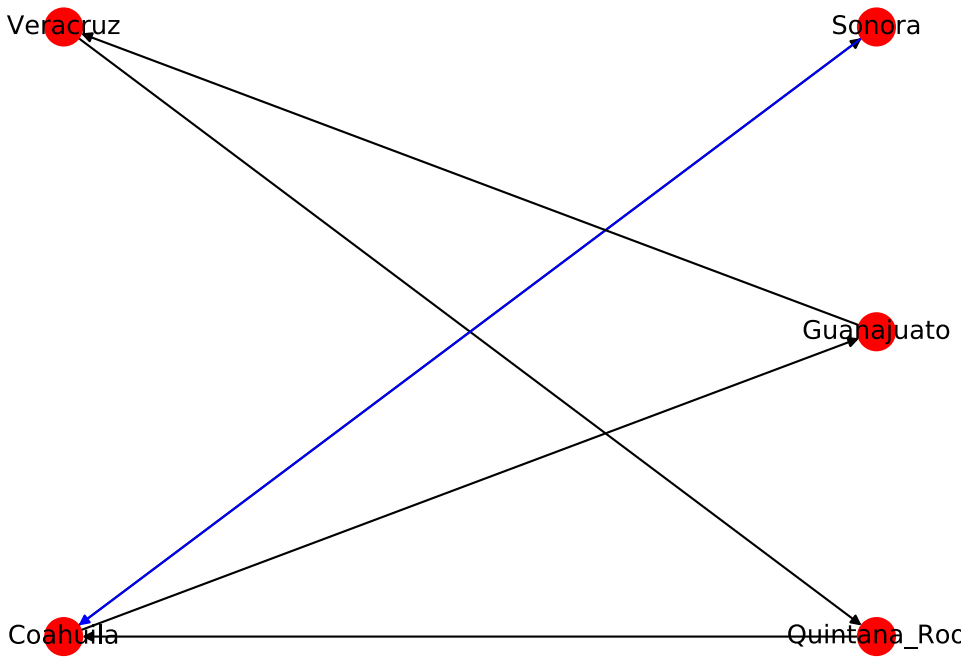


Figure 3: Bipartite layout

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

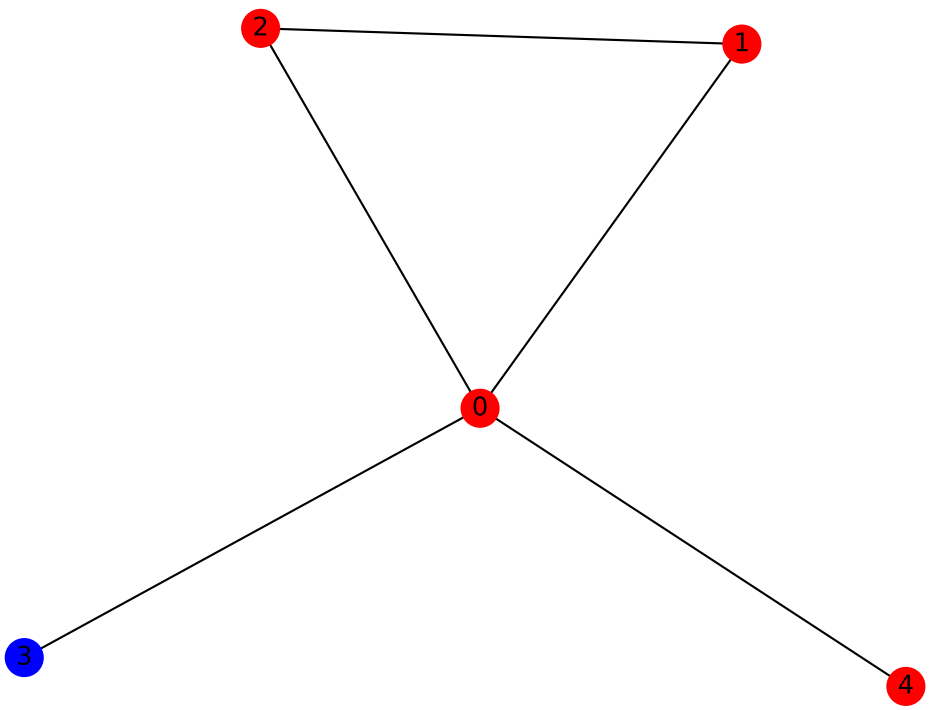


Figure 4: Kamada-Kawai layout for the undirected reflexive graph

Directed acyclic multigraph

```
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

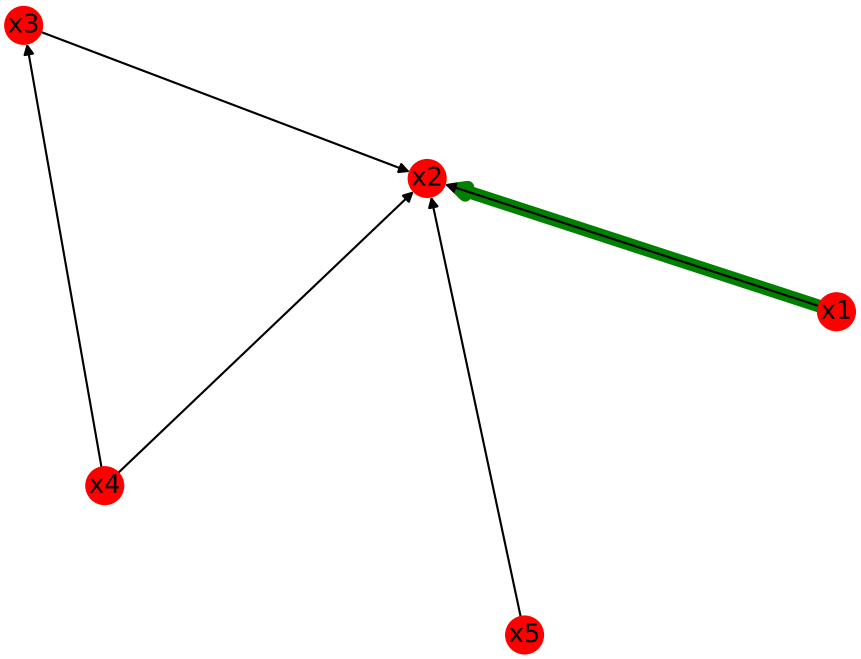


Figure 5: Kamada-Kawai layout for the directed acyclic multigraph

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

```python
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

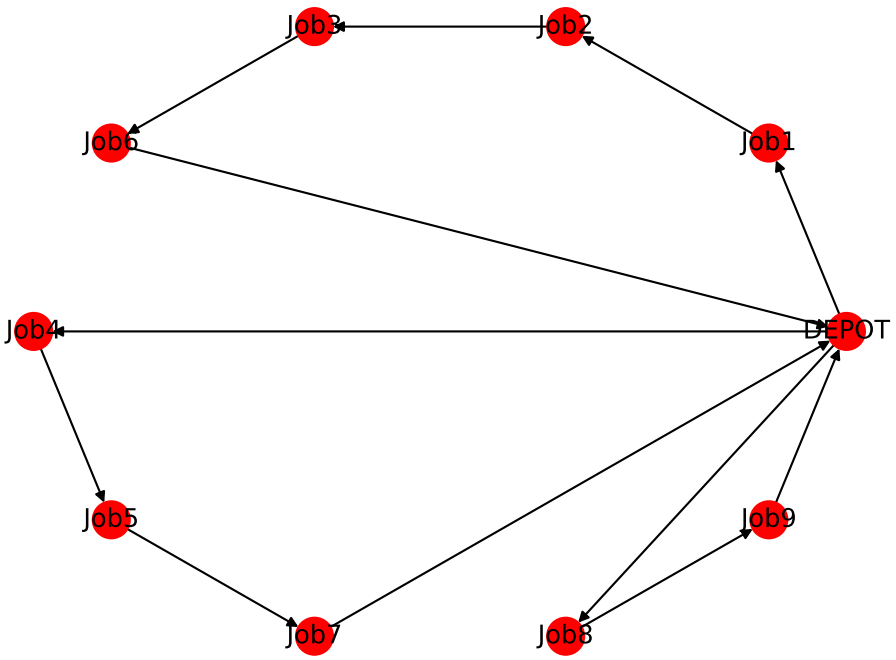


Figure 6: Shell layout for the directed cyclic graph

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

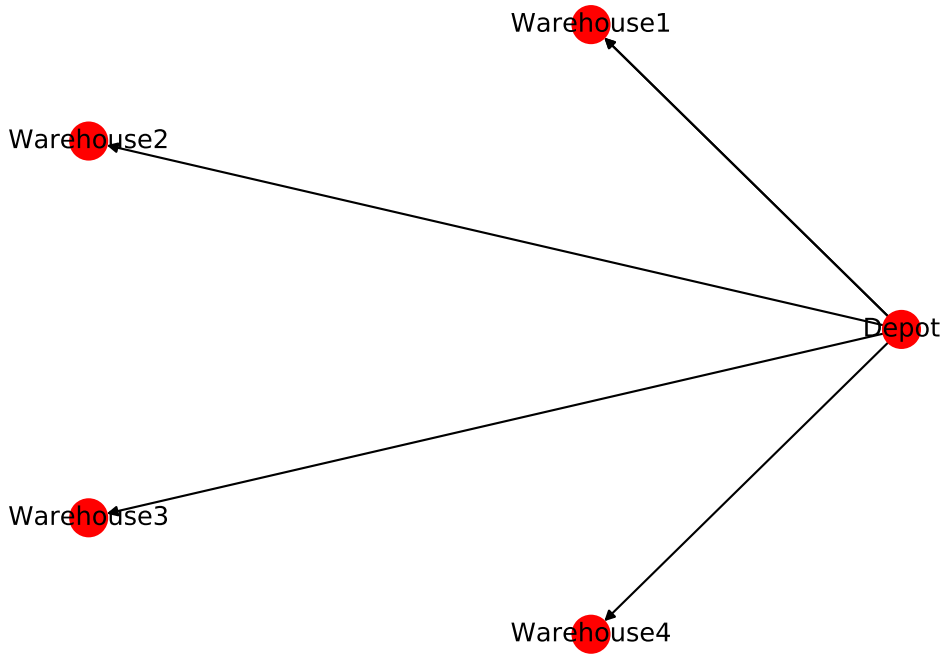


Figure 7: Shell layout for the directed cyclic graph

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

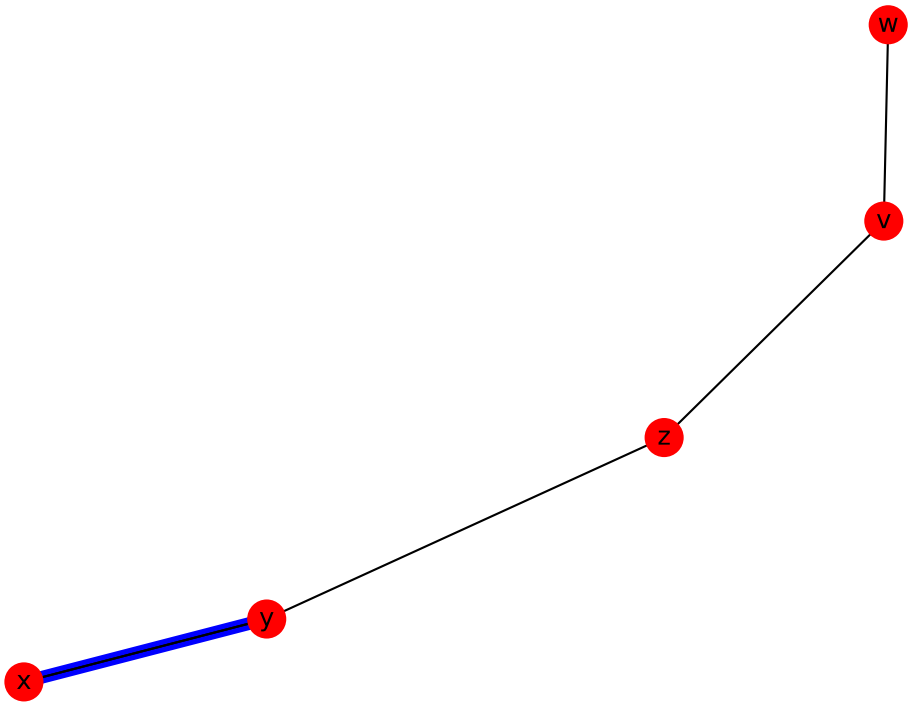


Figure 8: Fruchterman-Reingold layout for the undirected acyclic multigraph

Directed acyclic graph

```
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

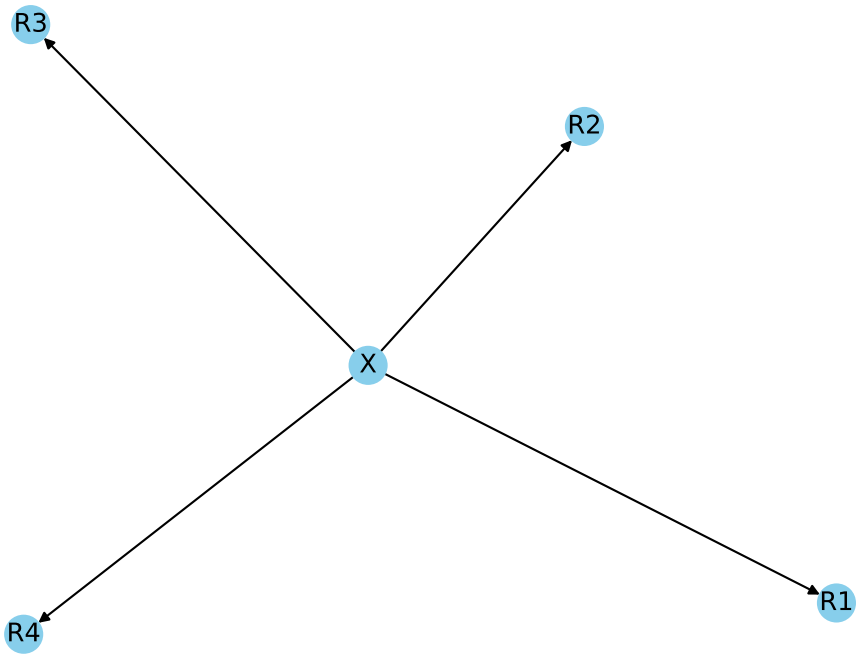


Figure 9: Fruchterman-Reingold layout for the directed acyclic graph

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].

```
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

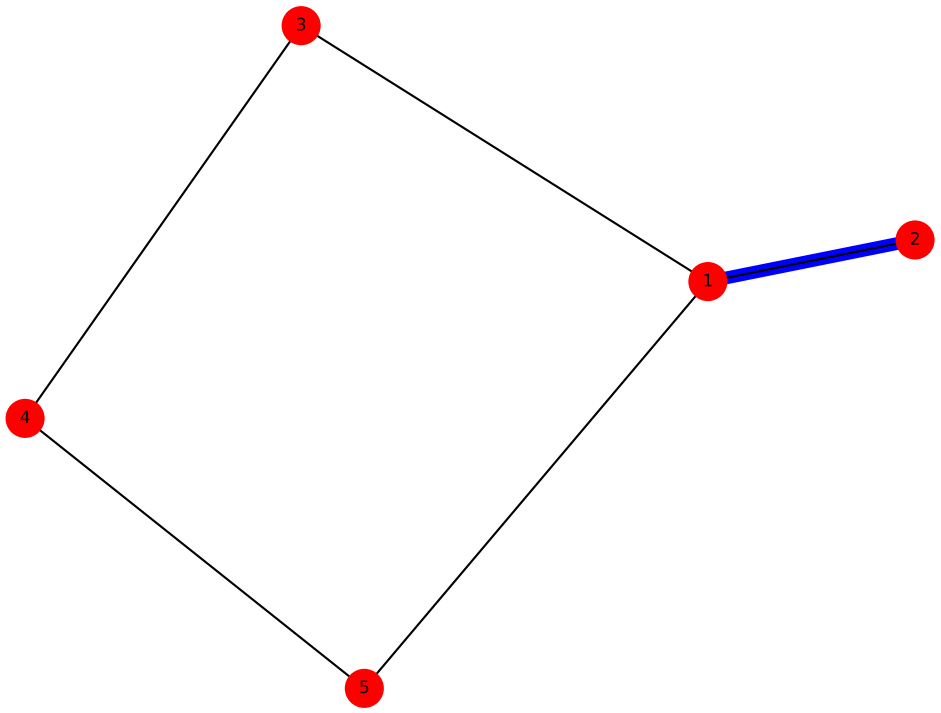


Figure 10: Spring layout

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 unidirected 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

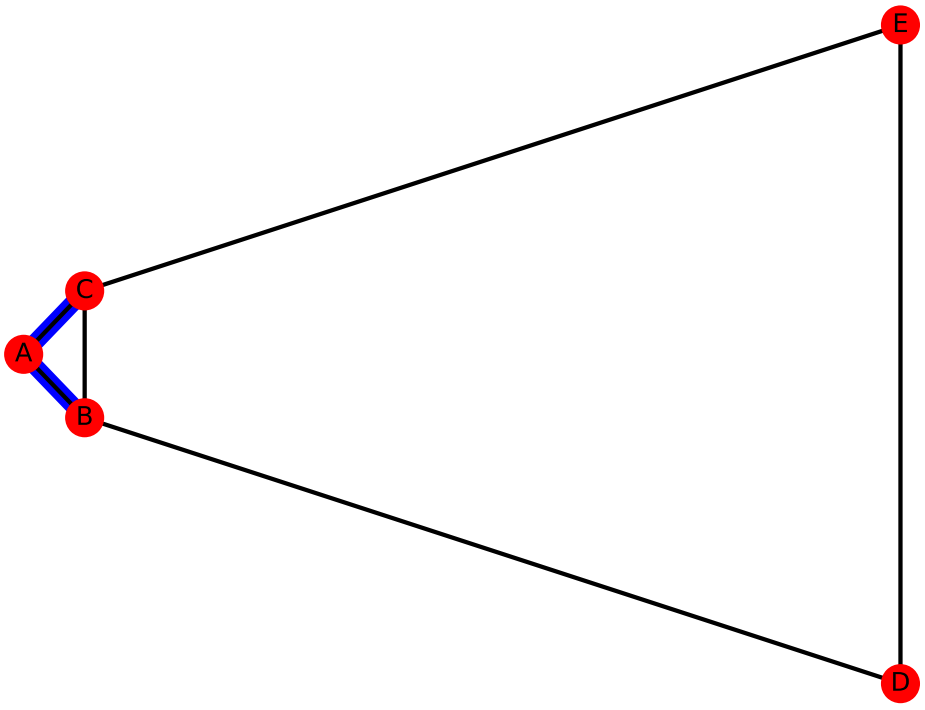


Figure 11: Spectral layout

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

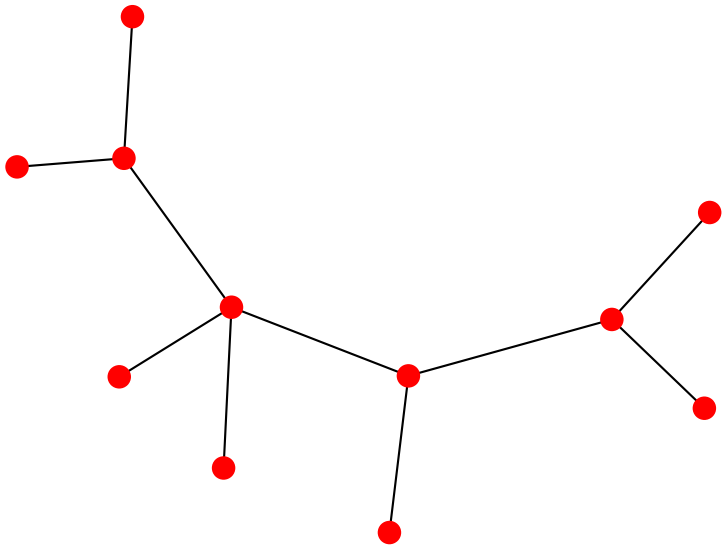


Figure 12: Force Atlas 2 layout

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.

Homework Assignment 2: Network Flows

5273

Circular Layout

As the name suggest this layout positions nodes in a circle [13]. According to Tsay et al. [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

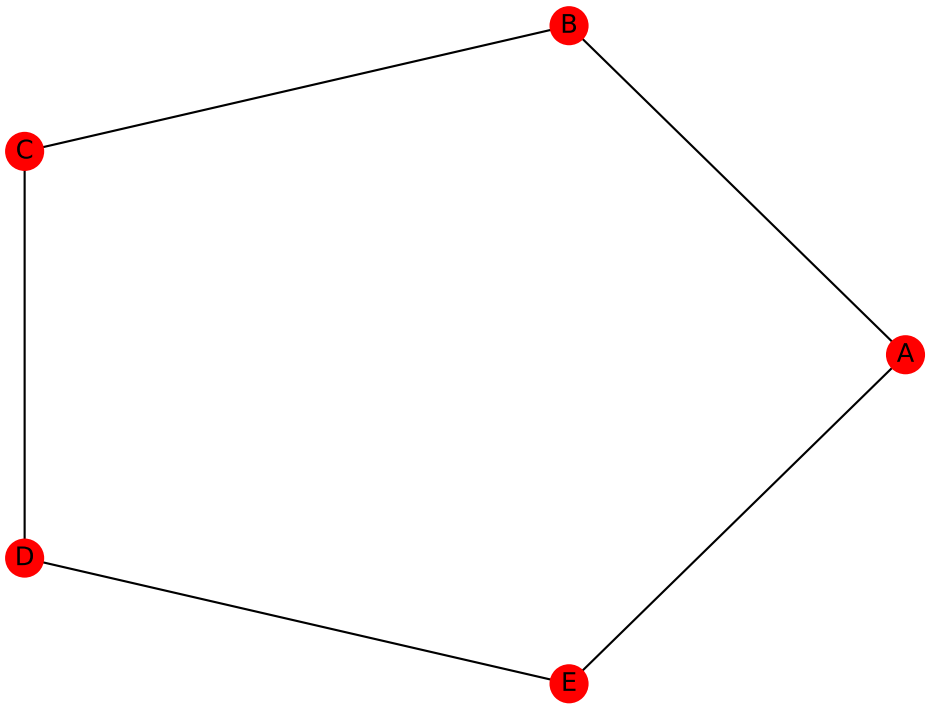


Figure 1: Circular layout

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 has the disadvantage of the difficulty in grasping the position of nodes [12].

```
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

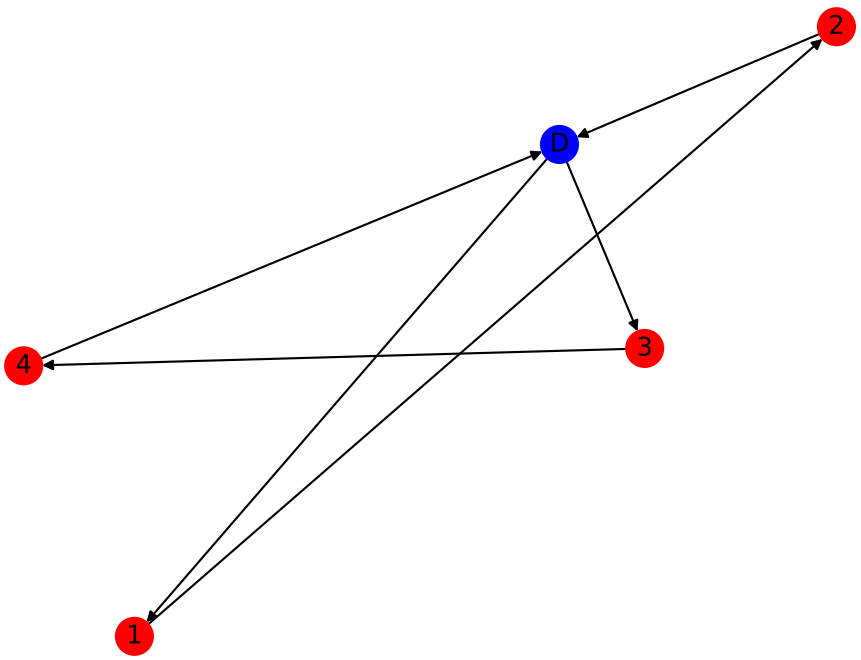


Figure 2: Random layout

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].

```
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

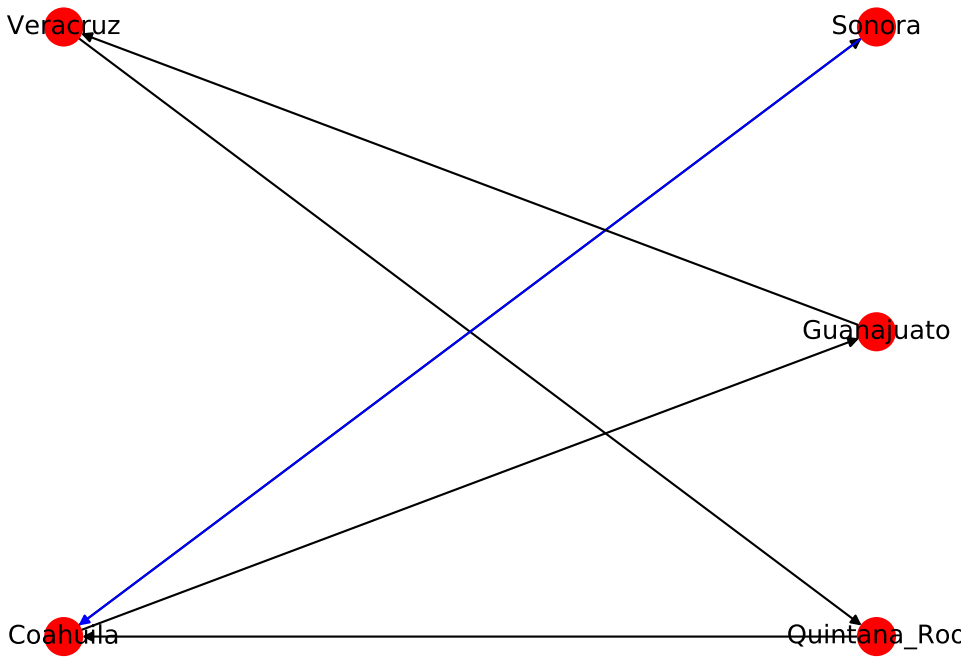


Figure 3: Bipartite layout

Kamada-Kawai Layout

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 classic 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

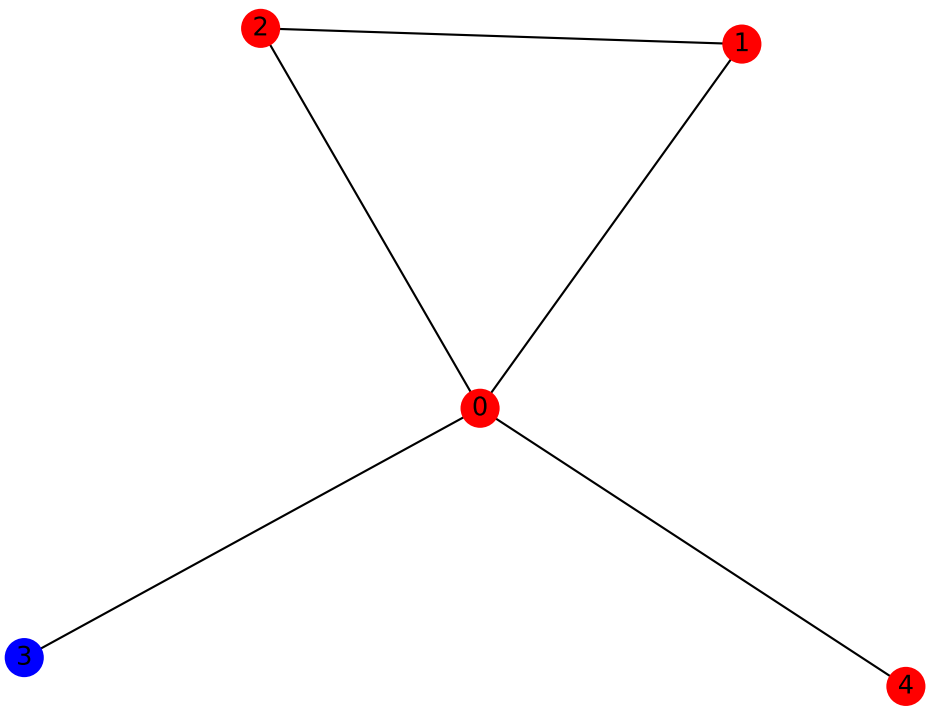


Figure 4: Kamada-Kawai layout for the undirected reflexive graph

Directed acyclic multigraph

```
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

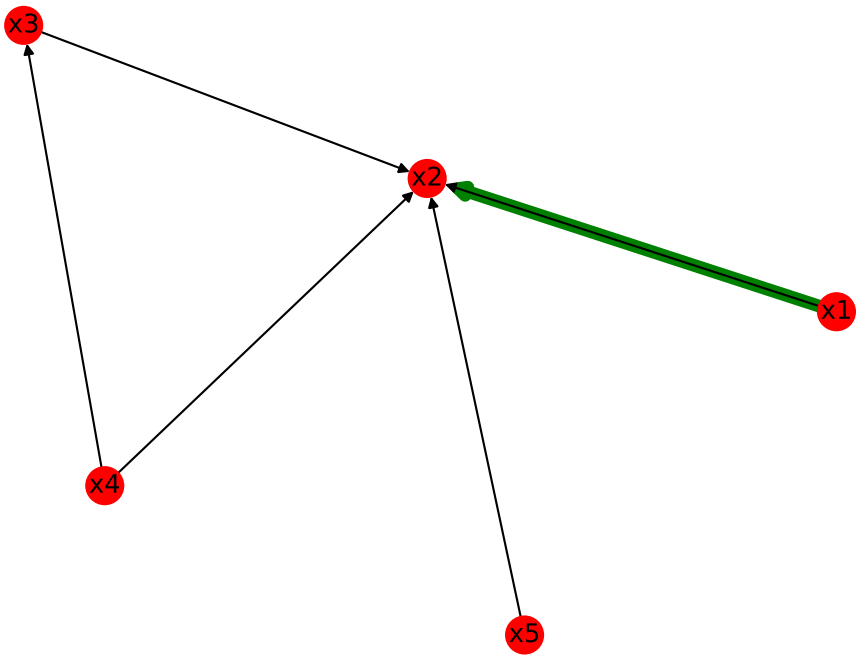


Figure 5: Kamada-Kawai layout for the directed acyclic multigraph

Shell Layout

This layout positions nodes in concentric circles [13]. It is also known as concentric layout [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 is 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('D')
G.add_nodes_from(['Job1','Job2','Job3','Job6'])
G.add_nodes_from(['Job4','Job5','Job7'])
G.add_nodes_from(['Job8','Job9'])

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

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

graph05dcg.py

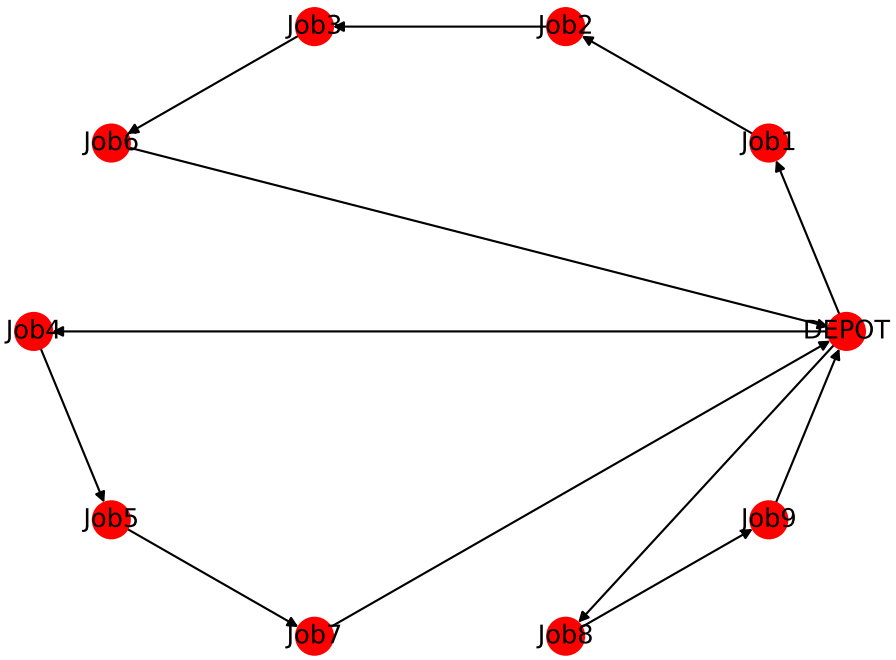


Figure 6: Shell layout for the directed cyclic graph

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

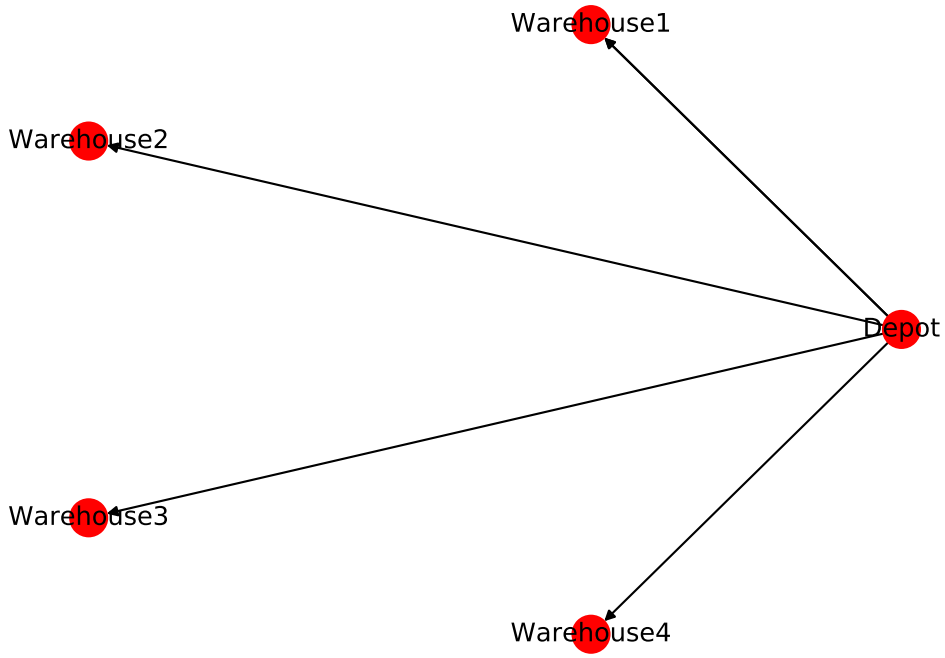


Figure 7: Shell layout for the directed cyclic graph

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

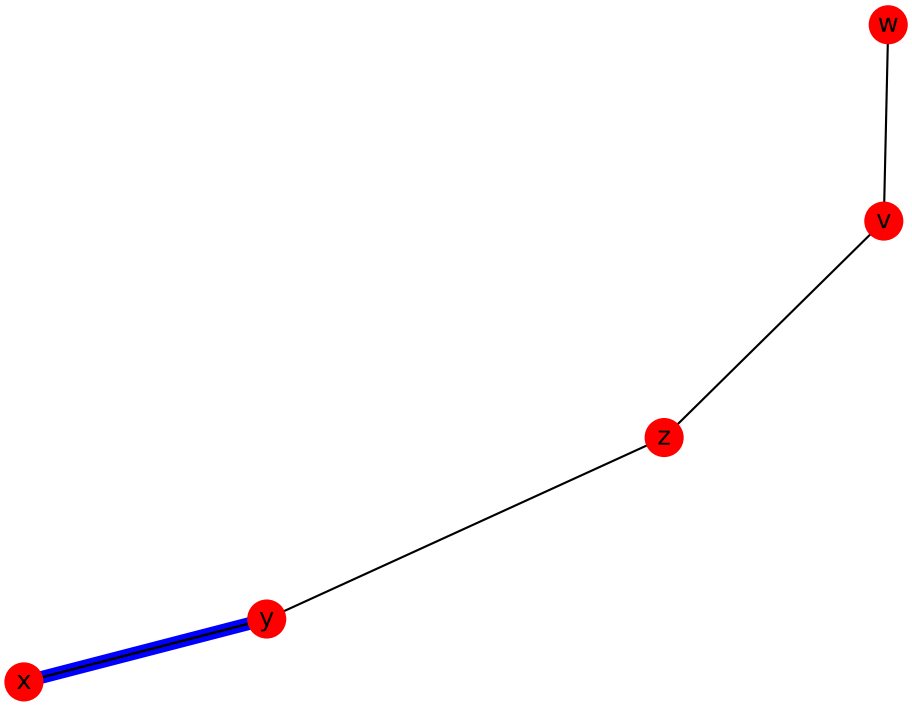


Figure 8: Fruchterman-Reingold layout for the undirected acyclic multigraph

Directed acyclic graph

```
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

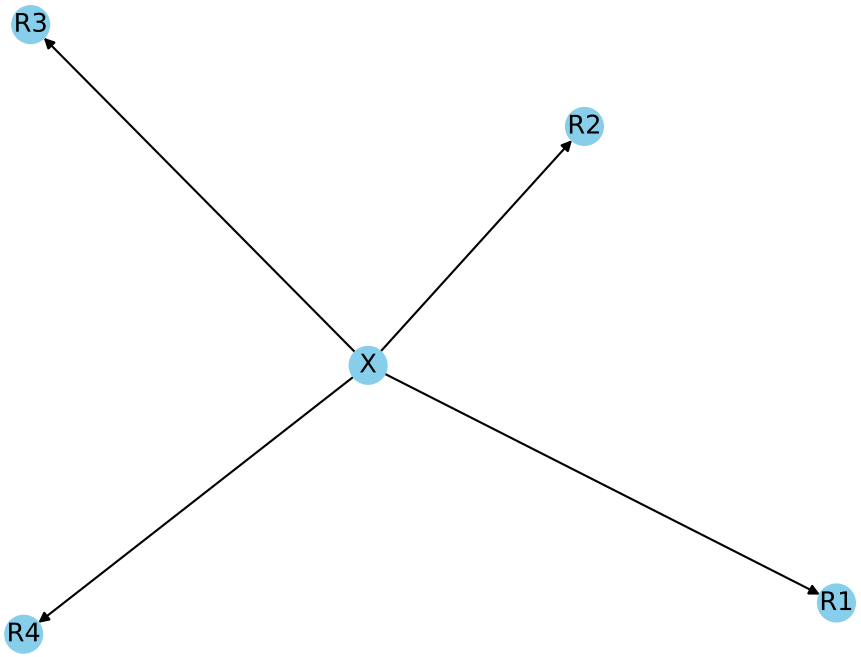


Figure 9: Fruchterman-Reingold layout for the directed acyclic graph

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 it is obtained a display with the possible symmetry [10].

```
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)

bw_centrality = nx.betweenness_centrality(M, normalized=False)
print(bw_centrality)

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

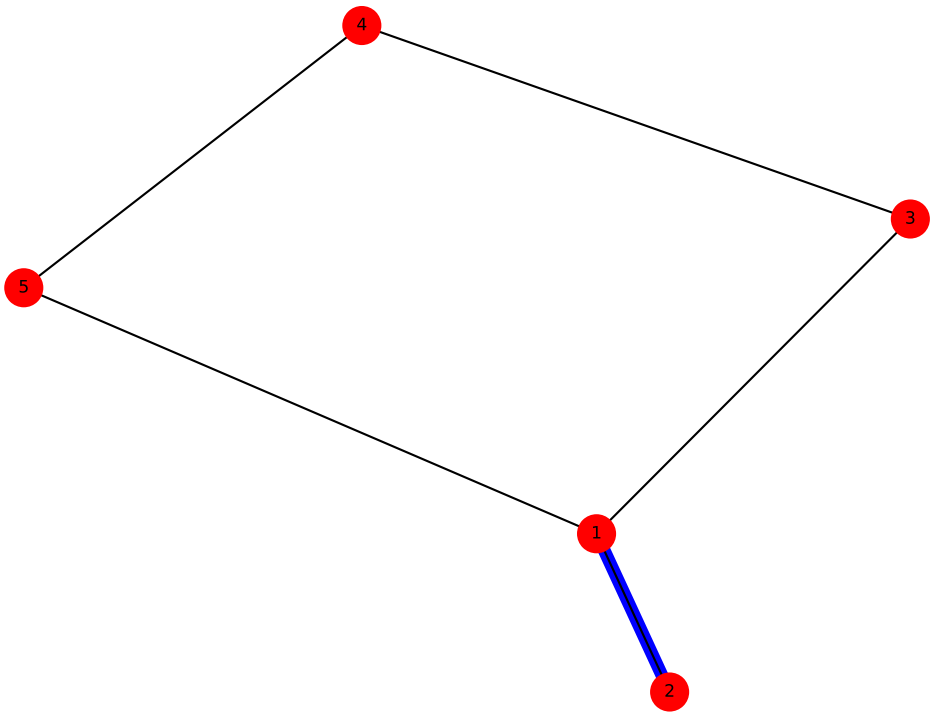


Figure 10: Spring layout

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. 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 undirected 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

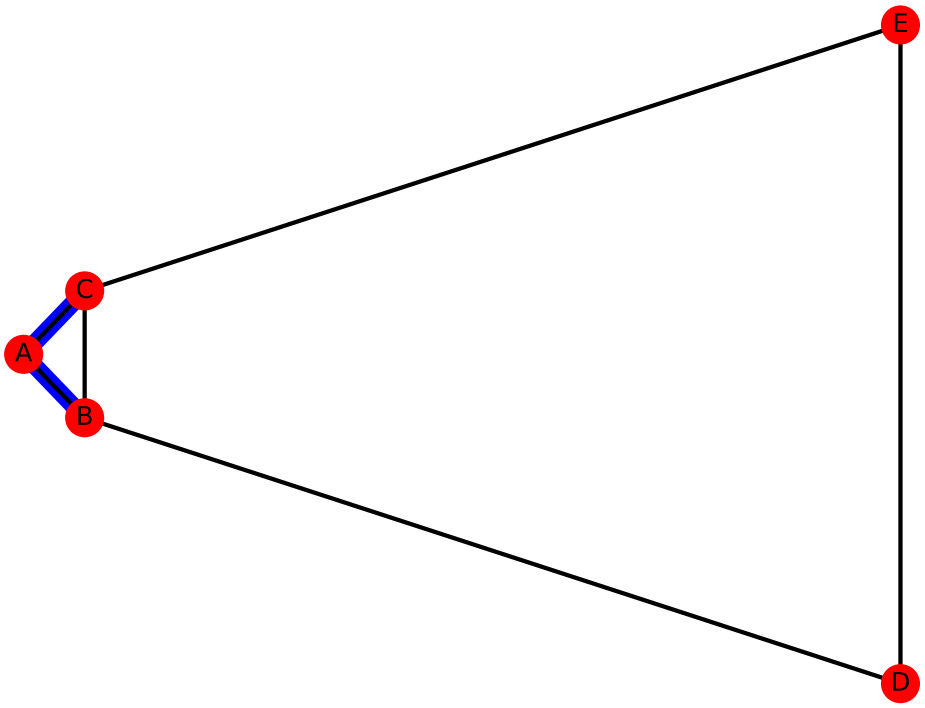


Figure 11: Spectral layout

Force Atlas 2 Layout

Force Atlas 2 is a very fast layout algorithm for force-directed graphs. It is 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.

```
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

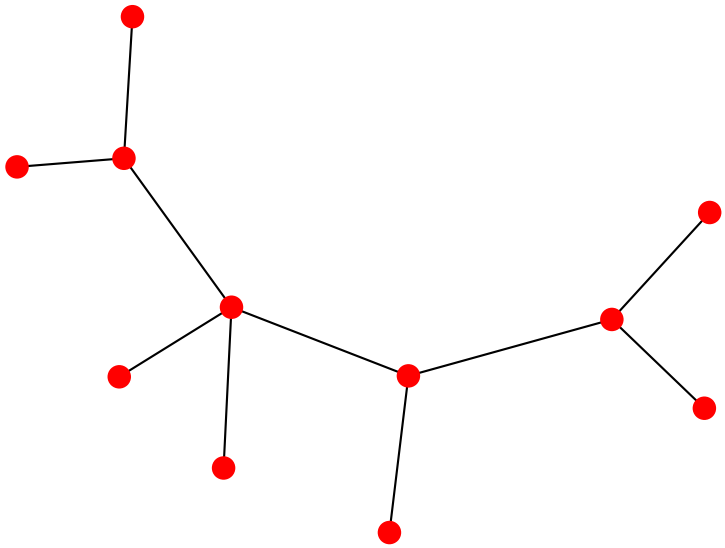


Figure 12: Force Atlas 2 layout

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. `http://data.opendataportal.at/dataset/bd825f93-cb2c-4f8c-836f-6d8ff2bf2a2c/resource/0b4ea5a8-d9f5-41be-ac85-0c88b4df3ca5/download/report.pdf`, Last accessed on 2019-02-12.

[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.

Homework 3 Corrections

In this homework corrections in notations and references were made, as well as in bibliography. The selected misused words were changed. Also punctuation marks were corrected as well as the order in which some phrases in the text appear.

Homework Assignment 3: Network Flows

5273

All Shortest Paths

Shortest Path Algorithms consist in finding a path between two nodes in a way that the sum of weights of its edges be minimum [9]. If a vertex has two or more predecessors, then there are two or more shortest paths, each of which must be followed separately if we wish to know all shortest paths from i to j [7].

```
P=nx.Graph()

P.add_nodes_from(['x1','x2','x3','x4','x5'])

P.add_edge('x1','x5',weight=1)
P.add_edge('x1','x4',weight=2)
P.add_edge('x4','x2',weight=2)
P.add_edge('x5','x2',weight=1)
P.add_edge('x2','x3',weight=8)
P.add_edge('x1','x2',weight=8)

pos=nx.kamada_kawai_layout(P)

replicas_asp=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        nx.all_shortest_paths(P,source='x1',target='x3',weight='weight')
    end=time.time()
    execution_time=end-start
    replicas_asp.append(execution_time)

normality_test=stats.shapiro(replicas_asp)
print(normality_test)
print(replicas_asp)

hist, bin_edges=np.histogram(replicas_asp,density=True)
first_edge, last_edge = np.min(replicas_asp),np.max(replicas_asp)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(asp,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.savefig("Histogram_asp.eps", format="EPS")
plt.show(1)
```

graphs.py

For the shortest path algorithm is selected an undirected graph with five nodes and six edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 0.0226 seconds with a standard deviation of 0.01161.
With the 30 values of the computation time a histogram is constructed.

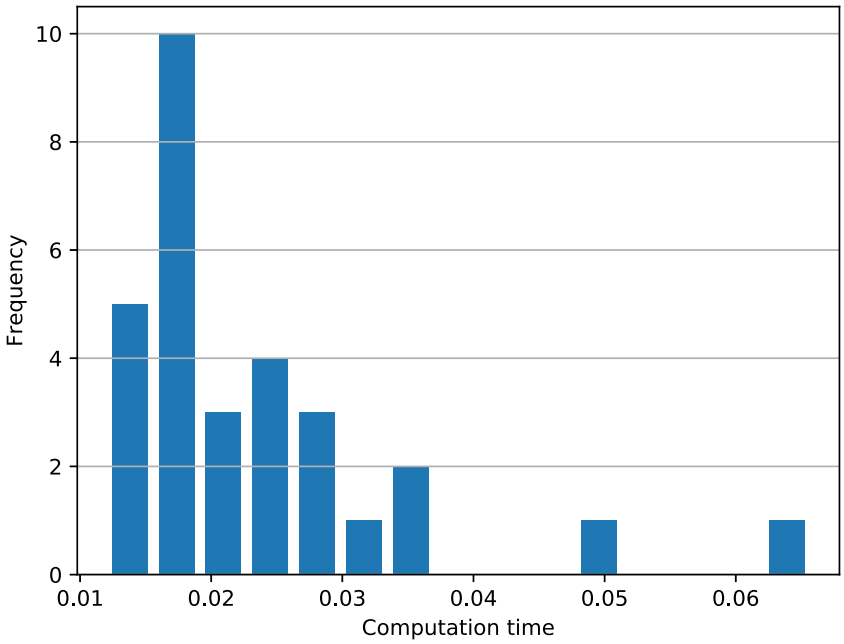


Figure 1: Histogram of All Shortest Paths Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 2.72×10^{-5} so the data is not normal distributed.

Betweenness Centrality

This algorithm measures the influence of a vertex over the flow of information in a graph based on shortest paths [10]. A node with higher betweenness centrality would have more control over the network, because more information will pass through that node [4].

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

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

replicas_bc=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        bw_centrality = nx.betweenness_centrality(B, normalized=False)
    end=time.time()
    execution_time=end-start
    replicas_bc.append(execution_time)

normality_test=stats.shapiro(replicas_bc)
print(normality_test)
print(replicas_bc)

hist, bin_edges=np.histogram(replicas_bc,density=True)
first_edge, last_edge = np.min(replicas_bc),np.max(replicas_bc)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_bc,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(2)
```

graphs.py

For the betweenness centrality algorithm is selected a directed graph with ten nodes and eleven edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 13.8352 seconds with a standard deviation of 0.4111.
With the 30 values of the computation time a histogram is constructed.

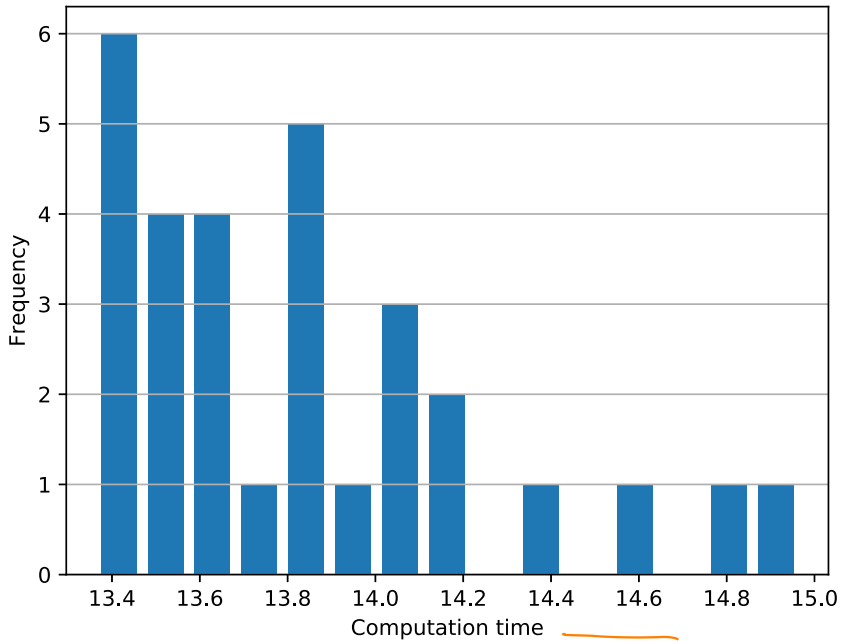


Figure 2: Histogram of Betweeness Centrality Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 0.004 so the data is not normal distributed.

Depth-First Search Tree

The Depth-First Search (DFS) has the application of traversing a graph and constructing a special structured tree, called a DFS tree [5]. A DFS tree of a graph is a spanning tree produced by performing a DFS algorithm in the graph. Starting from an arbitrary prescribed vertex , the DFS algorithm traverses the graph by repeatedly visiting an unvisited neighbor of the last visited vertex. If all the neighbors of the current vertex have already been visited, the search backtracks until it finds a vertex with an unvisited neighbor to continue [2].

```
D = nx.Graph()

D.add_node('Libro')
D.add_nodes_from(['C1','C2','C3'])
D.add_nodes_from(['s1.1','s1.2','s2.1','s2.2','s2.3'])
D.add_nodes_from(['s2.1.1','s2.1.2'])

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

replicas_dfs=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        dfs = nx.dfs_tree(D)
    end=time.time()
    execution_time=end-start
    replicas_dfs.append(execution_time)

normality_test=stats.shapiro(replicas_dfs)
print(normality_test)
print(replicas_dfs)

hist, bin_edges=np.histogram(replicas_dfs,density=True)
first_edge, last_edge = np.min(replicas_dfs),np.max(replicas_dfs)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_dfs,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(3)
```

graphs.py

For the DFS Tree algorithm is selected an undirected acyclic graph with eleven nodes and ten edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 2.5114 seconds with a standard deviation of 0.1115.
With the 30 values of the computation time a histogram is constructed.

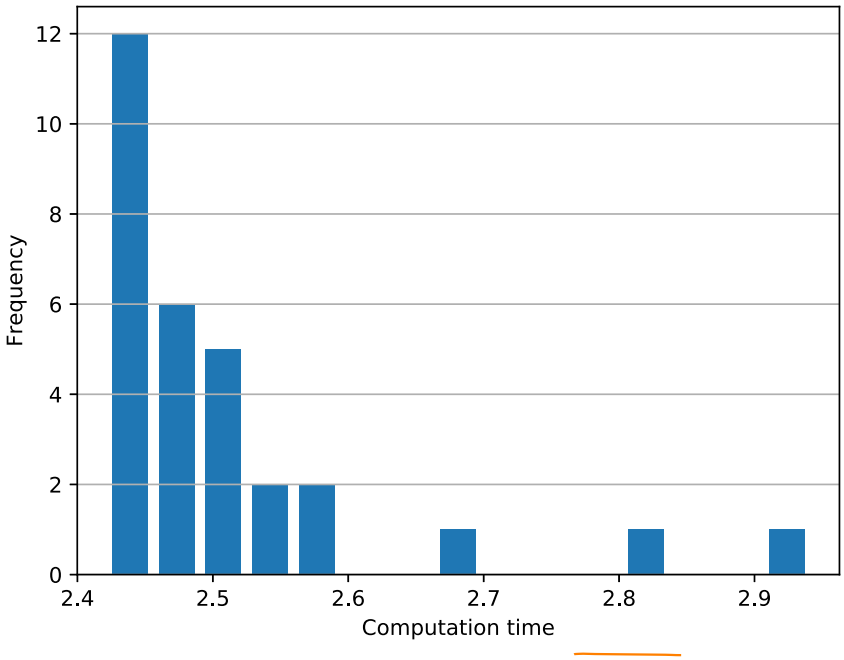


Figure 3: Histogram of DFS Tree Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 3.05×10^{-7} so the data is not normal distributed.

Strongly Connected Components

A strongly connected components of a directed graph is a maximal subset of vertices containing a directed path from each vertex to all others in the subset [3]. The traditional algorithm for finding the strongly connected components in a graph is based on depth first search [6].

```
S=nx.DiGraph()
S.add_nodes_from(['A','B','C','D','E'])

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

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

pos = nx.kamada_kawai_layout(S)

replicas_scc=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        sccs = list(nx.strongly_connected_components(S))
    end=time.time()
    execution_time=end-start
    replicas_scc.append(execution_time)

normality_test=stats.shapiro(replicas_scc)
print(normality_test)
print(replicas_scc)

hist, bin_edges=np.histogram(replicas_scc,density=True)
first_edge, last_edge = np.min(replicas_scc),np.max(replicas_scc)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_scc,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(4)
```

graphs.py

For the strongly connected components algorithm is selected a directed graph with five nodes and six edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 1.0484 seconds with a standard deviation of 0.0459.
With the 30 values of the computation time a histogram is constructed.

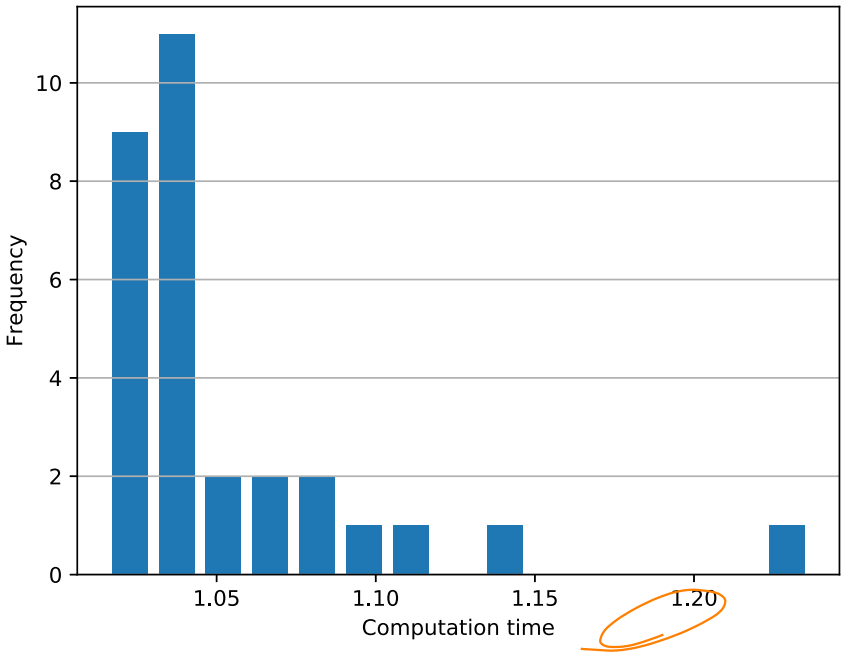


Figure 4: Histogram of Strongly Connected Components Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 1.05×10^{-6} so the data is not normal distributed.

Topological Sort

A topological sort is used to arrange the vertices of a directed acyclic graph in a linear order [8]. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks [1].

```
T = nx.DiGraph()

T.add_nodes_from(['r','t','u','v','w','x','y','z'])

T.add_edges_from([('x','y'),('r','t'),('t','v'),('t','u'),('u','w'),('x','z'),('y','z'),('y','v'),('v','w'),
                  ('v','x'),('x','z'),('z','v'),('z','w')],color='black',weight=1)

edges = T.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(T.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

pos = nx.shell_layout(T)

replicas_ts=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        ts = nx.topological_sort(T)
    end=time.time()
    execution_time=end-start
    replicas_ts.append(execution_time)

normality_test=stats.shapiro(replicas_ts)
print(normality_test)
print(replicas_ts)

hist, bin_edges=np.histogram(replicas_ts,density=True)
first_edge, last_edge = np.min(replicas_ts),np.max(replicas_ts)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_ts,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(5)
```

graphs.py

For the strongly connected components algorithm is selected a directed graph with five nodes and six edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 0.0136 seconds with a standard deviation of 0.003.
With the 30 values of the computation time a histogram is constructed.

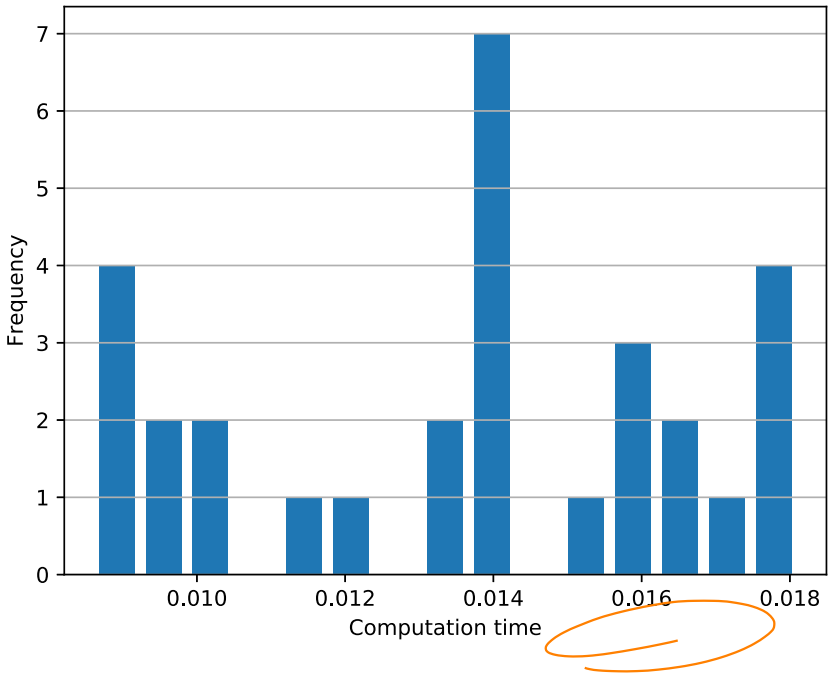
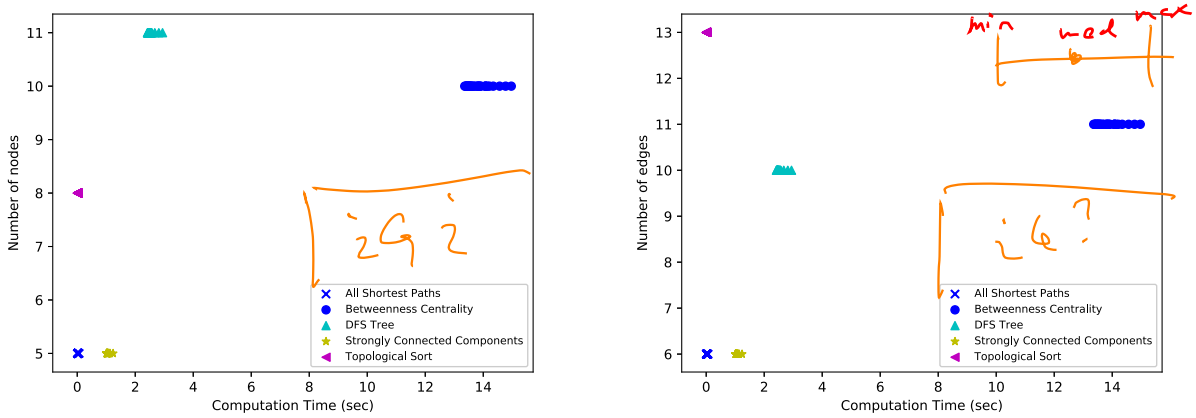


Figure 5: Histogram of Topological Sort Algorithm

Shapiro Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 0.0173 so the data is not normal distributed.

Conclutions

Once all algorithms are run, under the same conditions, we can conclude that none of the algorithms are normal distributed. To analyze all the data two scatterplot are also drawn. The first is of execution time vs. number of nodes, whereas the second one is of execution time vs. number of edges.



(a) Scatter plot of computation time vs. nodes

(b) Scatter plot of computation time vs. edges

Figure 6: Scatter plots.

From the analysis of scatterplots we can say that faster algorithms are: "All Shortest Path" and "Topological Sort".
A violin plot is also drawn, where the horizontal axis represents each of the algorithms and the vertical one represents the corresponding execution time. Here it is evident that the slowest algorithm is "Betweenness Centrality".

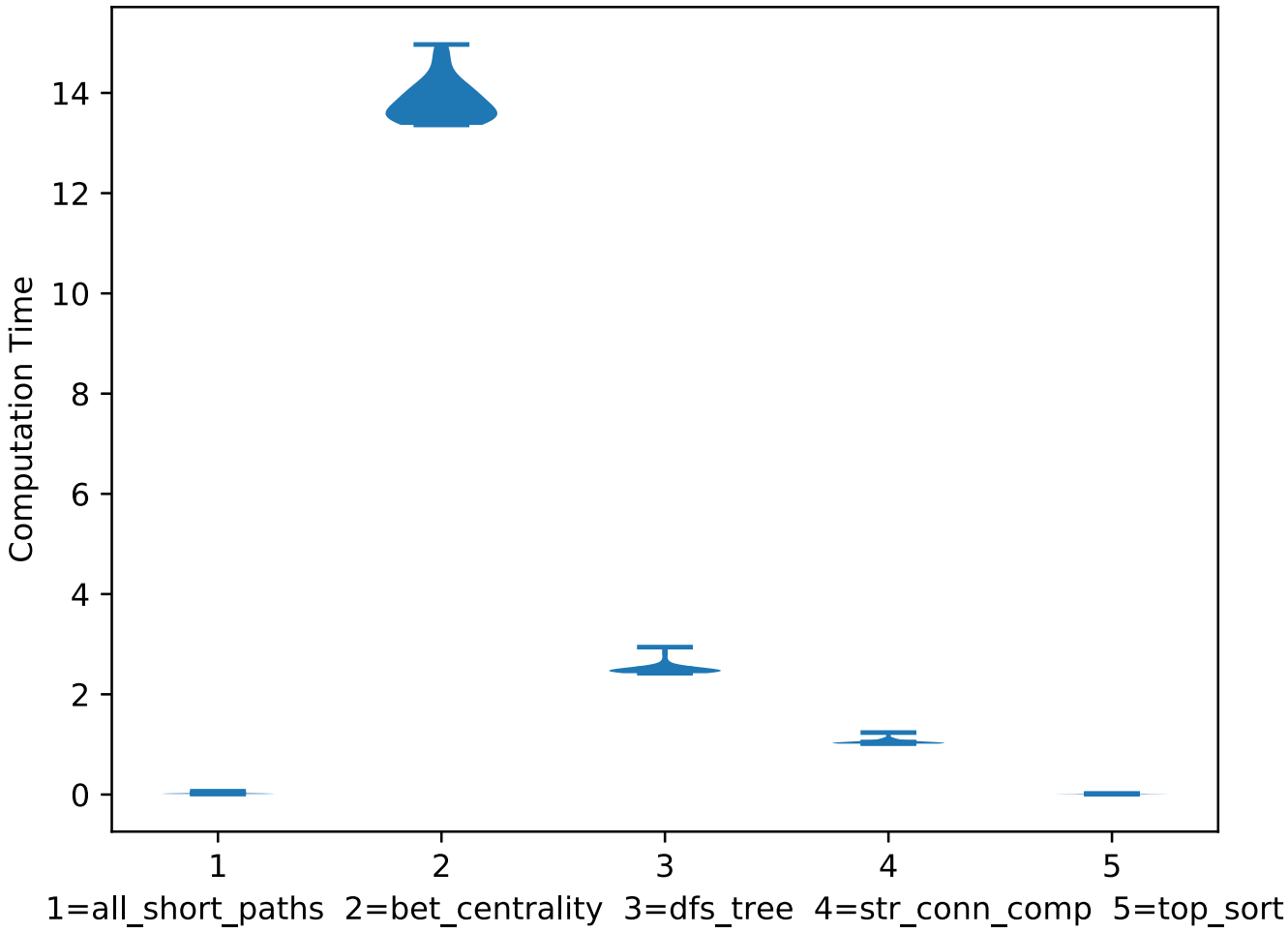


Figure 7: Violin Plot of Algorithms

References

[1] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal on computing*, 10(4):657–675, 1981.

[2] Amr Elmasry, Kurt Mehlhorn, and Jens Schmidt. Every dfs tree of a 3-connected graph contains a contractible edge. *Journal of Graph Theory*, 72(1):112–121, 2013.

[3] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, pages 505–511. Springer, 2000.

[4] Linton Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[5] Ephraim Korach and Zvi Ostfeld. Dfs tree construction: Algorithms and characterizations. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 87–106. Springer, 1988.

[6] William Mclendon, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[7] Mark Newman. Scientific collaboration networks, shortest paths, weighted networks, and centrality. *Physical Review*, 64(1):16–132, 2001.

[8] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics (JEA)*, 11:1–7, 2007.

[9] Jeanette Schmidt. All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings. In *Proceedings Third Israel Symposium on the Theory of Computing and Systems*, pages 67–77. IEEE, 1995.

[10] Raghavan Unnithan, Sunil Kumar, Balakrishnan Kannan, and Madambi Jathavedan. Betweenness centrality in some classes of graphs. *International Journal of Combinatorics*, pages 39–52, 2014.

Homework Assignment 3: Network Flows

5273

All Shortest Paths

Shortest Path Algorithms consist in finding a path between two nodes in a way that the sum of weights of its edges be minimum [9]. If a vertex has two or more predecessors, then there are two or more shortest paths, each of which must be followed separately if it is wished to know all shortest paths from i to j [7].

```
P=nx.Graph()

P.add_nodes_from(['x1','x2','x3','x4','x5'])

P.add_edge('x1','x5',weight=1)
P.add_edge('x1','x4',weight=2)
P.add_edge('x4','x2',weight=2)
P.add_edge('x5','x2',weight=1)
P.add_edge('x2','x3',weight=8)
P.add_edge('x1','x2',weight=8)

pos=nx.kamada_kawai_layout(P)

replicas_asp=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        nx.all_shortest_paths(P,source='x1',target='x3',weight='weight')
    end=time.time()
    execution_time=end-start
    replicas_asp.append(execution_time)

normality_test=stats.shapiro(replicas_asp)
print(normality_test)
print(replicas_asp)

hist, bin_edges=np.histogram(replicas_asp,density=True)
first_edge, last_edge = np.min(replicas_asp),np.max(replicas_asp)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_asp,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.savefig("Histogram_asp.eps", format="EPS")
plt.show(1)
```

graphs.py

For the shortest path algorithm an undirected graph with five nodes and six edges is selected. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 0.0226 seconds with a standard deviation of 0.01161.
With the 30 values of the computation time a histogram is constructed.

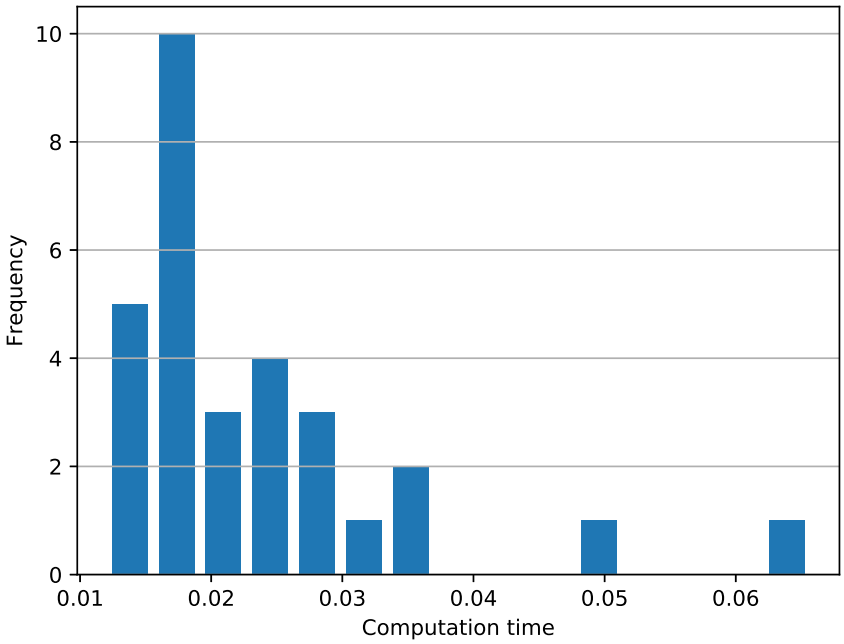


Figure 1: Histogram of All Shortest Paths Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 2.72×10^{-5} so the data is not normally distributed.

Betweenness Centrality

This algorithm measures the influence of a vertex over the flow of information in a graph based on shortest paths [10]. A node with higher betweenness centrality would have more control over the network, because more information will pass through that node [4].

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

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

replicas_bc=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        bw_centrality = nx.betweenness_centrality(B, normalized=False)
    end=time.time()
    execution_time=end-start
    replicas_bc.append(execution_time)

normality_test=stats.shapiro(replicas_bc)
print(normality_test)
print(replicas_bc)

hist, bin_edges=np.histogram(replicas_bc,density=True)
first_edge, last_edge = np.min(replicas_bc),np.max(replicas_bc)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_bc,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(2)
```

graphs.py

For the betweenness centrality algorithm a directed graph with ten nodes and eleven edges is selected. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 13.8352 seconds with a standard deviation of 0.4111.
With the 30 values of the computation time a histogram is constructed.

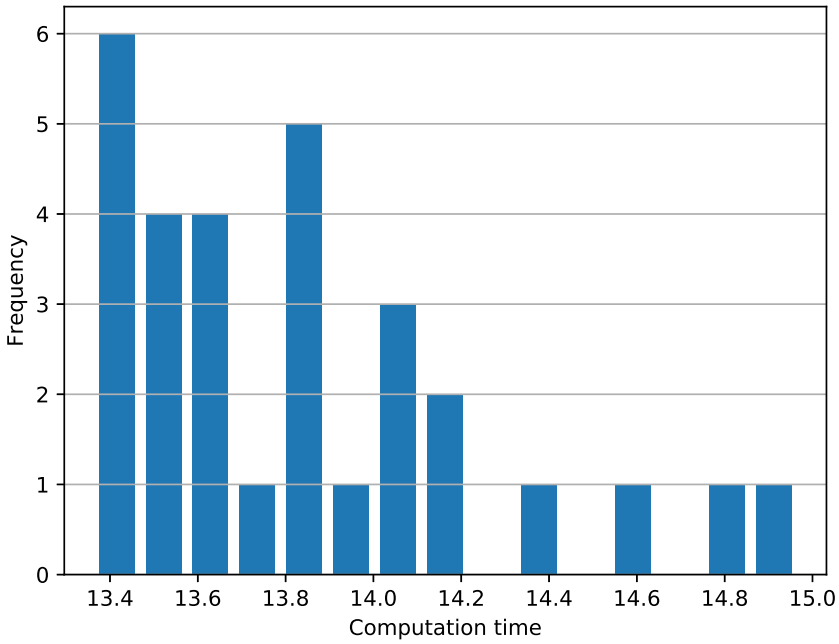


Figure 2: Histogram of Betweeness Centrality Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 0.004 so the data is not normally distributed.

Depth-First Search Tree

The Depth-First Search (DFS) has the application of traversing a graph and constructing a special structured tree, called a DFS tree [5]. A DFS tree of a graph is a spanning tree produced by performing a DFS algorithm in the graph. Starting from an arbitrary prescribed vertex, the DFS algorithm traverses the graph by repeatedly visiting an unvisited neighbor of the last visited vertex. If all the neighbors of the current vertex have already been visited, the search backtracks until it finds a vertex with an unvisited neighbor to continue [2].

```
D = nx.Graph()

D.add_node('Libro')
D.add_nodes_from(['C1','C2','C3'])
D.add_nodes_from(['s1.1','s1.2','s2.1','s2.2','s2.3'])
D.add_nodes_from(['s2.1.1','s2.1.2'])

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

replicas_dfs=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        dfs = nx.dfs_tree(D)
    end=time.time()
    execution_time=end-start
    replicas_dfs.append(execution_time)

normality_test=stats.shapiro(replicas_dfs)
print(normality_test)
print(replicas_dfs)

hist, bin_edges=np.histogram(replicas_dfs,density=True)
first_edge, last_edge = np.min(replicas_dfs),np.max(replicas_dfs)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_dfs,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(3)
```

graphs.py

For the DFS Tree algorithm an undirected acyclic graph with eleven nodes and ten edges is selected. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 2.5114 seconds with a standard deviation of 0.1115.
With the 30 values of the computation time a histogram is constructed.

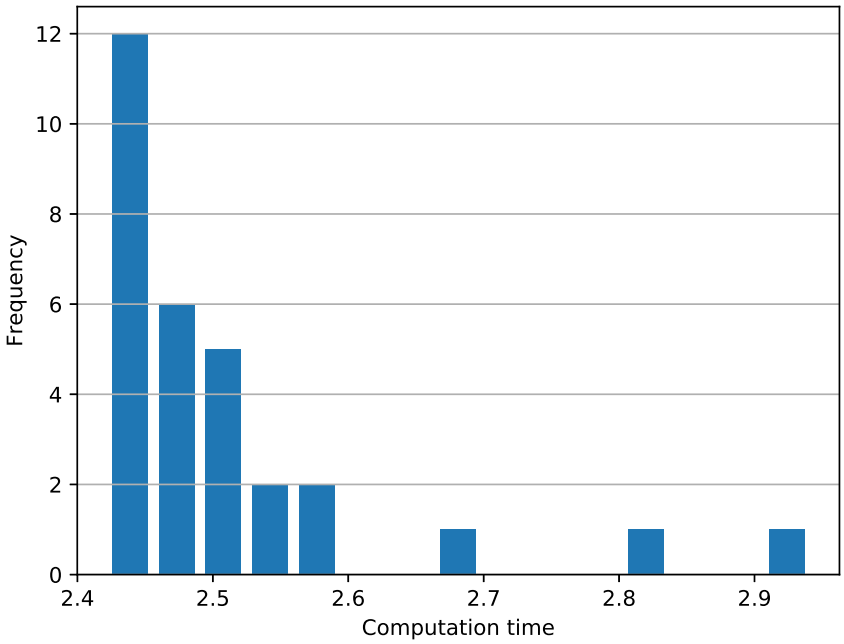


Figure 3: Histogram of DFS Tree Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 3.05×10^{-7} so the data is not normally distributed.

Strongly Connected Components

A strongly connected components of a directed graph is a maximal subset of vertices containing a directed path from each vertex to all others in the subset [3]. The traditional algorithm for finding the strongly connected components in a graph is based on depth first search [6].

```
S=nx.DiGraph()
S.add_nodes_from(['A','B','C','D','E'])

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

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

pos = nx.kamada_kawai_layout(S)

replicas_scc=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        sccs = list(nx.strongly_connected_components(S))
    end=time.time()
    execution_time=end-start
    replicas_scc.append(execution_time)

normality_test=stats.shapiro(replicas_scc)
print(normality_test)
print(replicas_scc)

hist, bin_edges=np.histogram(replicas_scc,density=True)
first_edge, last_edge = np.min(replicas_scc),np.max(replicas_scc)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_scc,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(4)
```

graphs.py

For the strongly connected components algorithm a directed graph with five nodes and six edges is selected. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 1.0484 seconds with a standard deviation of 0.0459.
With the 30 values of the computation time a histogram is constructed.

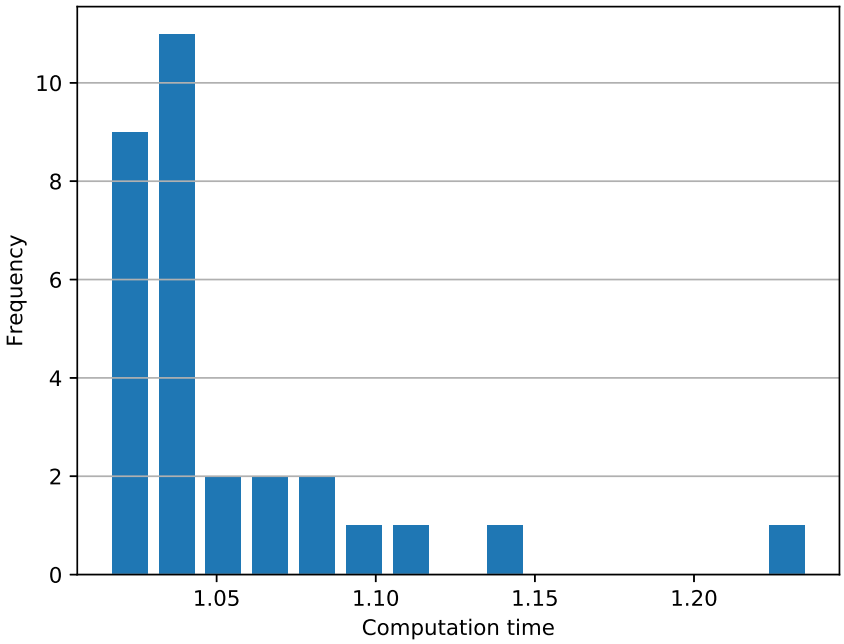


Figure 4: Histogram of Strongly Connected Components Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 1.05×10^{-6} so the data is not normally distributed.

Topological Sort

A topological sort is used to arrange the vertices of a directed acyclic graph in a linear order [8]. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is a valid sequence for the tasks [1].

```
T = nx.DiGraph()

T.add_nodes_from(['r','t','u','v','w','x','y','z'])

T.add_edges_from([('x','y'),('r','t'),('t','v'),('t','u'),('u','w'),('x','z'),('y','z'),('y','v'),('v','w'),
                  ('v','x'),('x','z'),('z','v'),('z','w')],color='black',weight=1)

edges = T.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(T.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

pos = nx.shell_layout(T)

replicas_ts=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        ts = nx.topological_sort(T)
    end=time.time()
    execution_time=end-start
    replicas_ts.append(execution_time)

normality_test=stats.shapiro(replicas_ts)
print(normality_test)
print(replicas_ts)

hist, bin_edges=np.histogram(replicas_ts,density=True)
first_edge, last_edge = np.min(replicas_ts),np.max(replicas_ts)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_ts,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(5)
```

graphs.py

For the strongly connected components algorithm a directed graph with five nodes and six edges is selected. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 0.0136 seconds with a standard deviation of 0.003.
With the 30 values of the computation time a histogram is constructed.

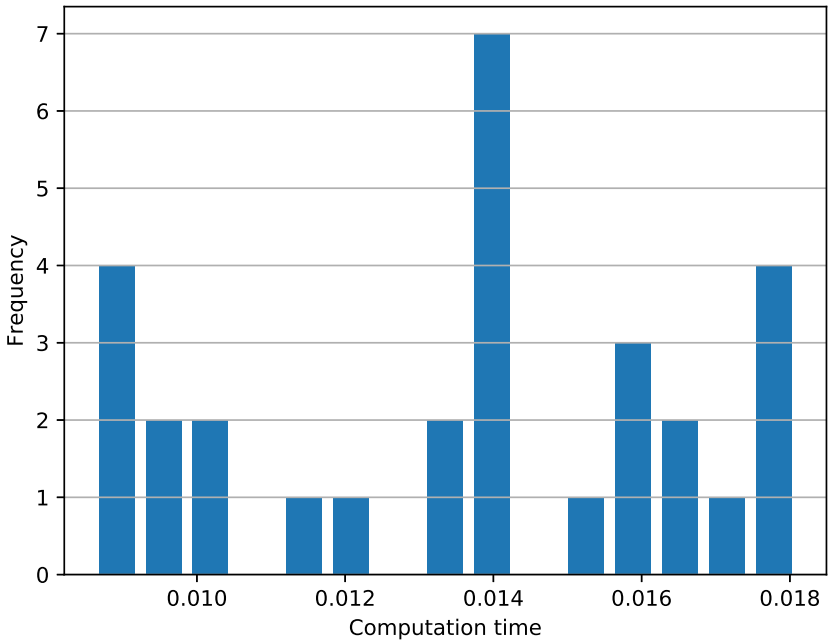
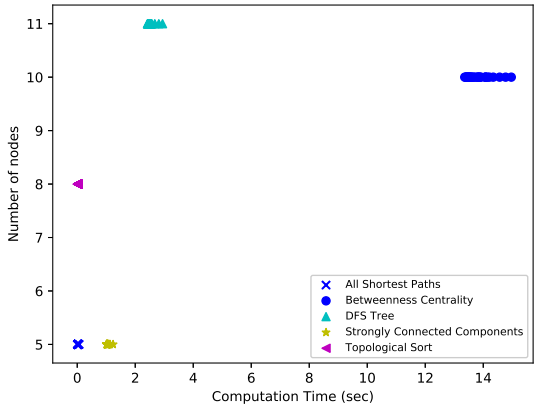


Figure 5: Histogram of Topological Sort Algorithm

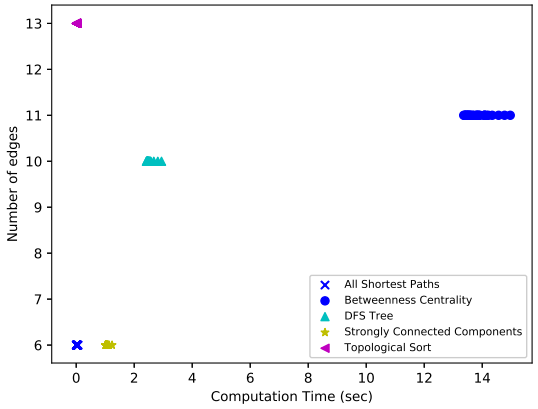
Shapiro Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 0.0173 so the data is not normally distributed.

Conclutions

Once all algorithms are run, under the same conditions, it is conclude that no algorithm is normally distributed. To analyze all the data two scatterplot are also drawn. The first is of execution time vs. number of nodes, whereas the second one is of execution time vs. number of edges.



(a) Scatter plot of computation time vs. nodes



(b) Scatter plot of computation time vs. edges

Figure 6: Scatter plots.

From the analysis of scatterplots we can say that faster algorithms are: “All Shortest Path” and “Topological Sort”.
A violin plot is also drawn, where the horizontal axis represents each of the algorithms and the vertical one represents the corresponding execution time. Here it is evident that the slowest algorithm is “Betweenness Centrality”.

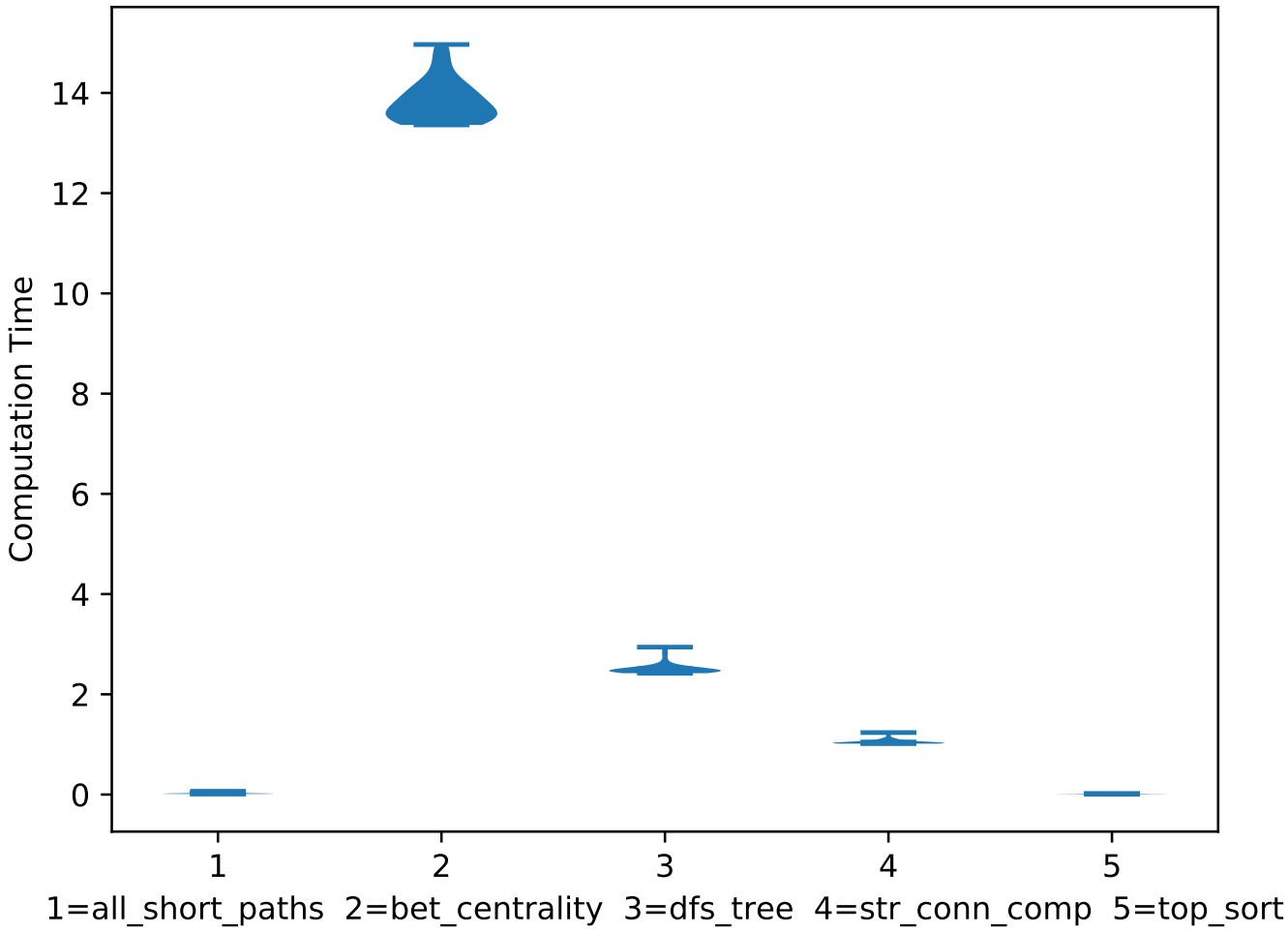


Figure 7: Violin Plot of Algorithms

References

[1] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal on computing*, 10(4):657–675, 1981.

[2] Amr Elmasry, Kurt Mehlhorn, and Jens Schmidt. Every DFS tree of a 3-connected graph contains a contractible edge. *Journal of Graph Theory*, 72(1):112–121, 2013.

[3] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, pages 505–511. Springer, 2000.

[4] Linton Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[5] Ephraim Korach and Zvi Ostfeld. DFS tree construction: Algorithms and characterizations. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 87–106. Springer, 1988.

[6] William McLendon, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[7] Mark Newman. Scientific collaboration networks, shortest paths, weighted networks, and centrality. *Physical Review*, 64(1):16–132, 2001.

[8] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics*, 11:1–7, 2007.

[9] Jeanette Schmidt. All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings. In *Proceedings Third Israel Symposium on the Theory of Computing and Systems*, pages 67–77. IEEE, 1995.

[10] Raghavan Unnithan, Sunil Kumar, Balakrishnan Kannan, and Madambi Jathavedan. Betweenness centrality in some classes of graphs. *International Journal of Combinatorics*, pages 39–52, 2014.

Homework 4 Corrections

In this homework corrections in notations were made. The selected misused words in the feedback were changed. Also punctuation marks were corrected as well as the order in which some phrases in the text appear. Arrengements in the listing package and tables were done in order to show the Python code and the text file “ANOVA” in a better way.

Homework Assignment 4: Network Flows

5273

Introduction

For this work graph generators of the library NetworkX of Python have been used. There have been selected three of these generator for the study: Complete Graph, Wheel Graph and Star Graph. To use each one of them there have been selected a base two as logarithmic base to generate four different graph orders, so the total number of nodes for each graph generator is 4, 8, 16 and 32 nodes respectively. The total number of edges depends on the chosen graph. Edge weights have been generated following a normally distribution with mean of 12 units and a standard deviation of 0.4.
After the assigment of normally distributed weights to edges, these weights would be used as instances of the maximum flow problem. Three implementations of this problem are performed using NetworkX algorithms: Maximum Flow, Minimum Cut and Preflow Push. Each one implementations are performed for each one of the graph generators.
This work it is run on an *Intel*® Celeron CPU @ 1.10 GHz with 4 GB RAM Laptop.

Graph Generators

Complete Graph

The first graph generator chosen is the complete graph. This is a simple undirected graph which each pair of nodes is connected by a unique edge [1]. The complete graphs selected, as mentioned at Introduction, have 4, 8, 16 and 32 nodes with 6, 20, 120 and 496 edges respectively.

Wheel Graph

The second graph generator chosen is the wheel graph. This is a simple undirected graph characterized by its single hub node connected to each of the $(n-1)$-node cycle graph [5]. The wheel graphs selected, as well as the other graph generators, have 4, 8, 16 and 32 nodes with 6, 14, 30 and 462 edges respectively.

Star Graph

The third graph generator chosen is the star graph. This is a simple undirected graph that it can be considered as a tree with one internal node and k leaves [4]. The star graphs selected, as well as the other graph generators, have an order of 4, 8, 16 and 32 nodes with also 4, 8, 16 and 32 edges respectively.

```
for j in range(10):
    for i in range(2,6):
        i = 2 ** i
        number_of_nodes.append(i)

        #COMPLETE GRAPH
        start_C = time.time()
        C=nx.complete_graph(i)
        end_C = time.time()
        execution_time_C = end_C - start_C
        complete_graph_times.append(execution_time_C)

        assign_normally_distributed_weight(12,0.4,C)

        # WHEEL GRAPH
        start_W = time.time()
        W = nx.wheel_graph(i)
        end_W = time.time()
        execution_time_W = end_W - start_W
        wheel_graph_times.append(execution_time_W)

        assign_normally_distributed_weight(12, 0.4, W)

        # STAR GRAPH
        start_S = time.time()
        S = nx.star_graph(i)

        end_S = time.time()
        execution_time_S = end_S - start_S
        star_graph_times.append(execution_time_S)

        assign_normally_distributed_weight(12, 0.4, S)
```

graphs_t4.py

Maximum Flow Algorithms

Maximum Flow

The algorithm can be used in the three chosen graph generators since the settings of the arc capacities are not fixed and are functions of a single parameter [2], it have been defined the source and the sink in each graph and a positive capacity for every edge. The value of the flow is the net flow out of the source. The maximum flow problem is that of finding a flow of maximum value [3].

```
        #Maximum flow Complete Graph
        start_flow = time.time()
        for k in range(5):
            flow_value = nx.maximum_flow(C, ss_list[k][0], ss_list[k][1], capacity='weight
')
        end_flow = time.time()
        execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
        # Maximum flow Wheel Graph
        start_flow = time.time()
        for k in range(5):
            flow_value = nx.maximum_flow(W, ss_list[k][0], ss_list[k][1], capacity='weight
')
        end_flow = time.time()
        execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Maximum flow Star Graph
            start_flow = time.time()
            for k in range(5):
                flow_value = nx.maximum_flow(S, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

Minimum Cut

This algorithm performed well with the three chosen graph generators, which are undirected edge-weighted graphs. The minimum cut of an undirected graph with edge weights consists in a set of edges with minimum sum of weights, such that its removal would cause the graph to become disconnected [6].

```
            #Minimum cut Complete Graph
            start_flow = time.time()
            for k in range(5):
                cut_value = nx.minimum_cut(C, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Minimum cut Wheel Graph
            start_flow = time.time()
            for k in range(5):
                cut_value = nx.minimum_cut(W, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Minimum cut Star Graph
            start_flow = time.time()
            for k in range(5):
                cut_value = nx.minimum_cut(S, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

Preflow Push

This implementation finds a maximum single-commodity flow using the highest-label preflow-push algorithm [5] . It returns the residual network resulting after computing the maximum flow, which also have been performed.

```
            #Preflow push Complete Graph
            start_flow = time.time()
            for k in range(5):
                R = preflow_push(C, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Preflow push Wheel Graph
            start_flow = time.time()
            for k in range(5):
                R = preflow_push(W, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Preflow push Star Graph
            start_flow = time.time()
            for k in range(5):
                R = preflow_push(S, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
            preflow_push_times.append(execution_time_flow)
```

graphs_t4.py

Experimental Results

Once all corresponding graphs are generated it can be evaluated throughout statistical methods and visualizations the effect of the used graph generator in the computation time, as well as the maximum flow algorithms, the graph order and the density of the graph.
For every effect a boxplot is shown.
The first boxplot shows the effect of the graph generator in the computation time.
Data for Complete Graph has a mean of 0.01[illegible] and a standard deviation of 0.013[illegible], whereas data for Wheel Graph has a mean of 0.002[illegible] and a standart deviation of 0.001[illegible] and the Star Graph Generator has a mean of 0.00[illegible] and a standard deviation of 0.002[illegible]

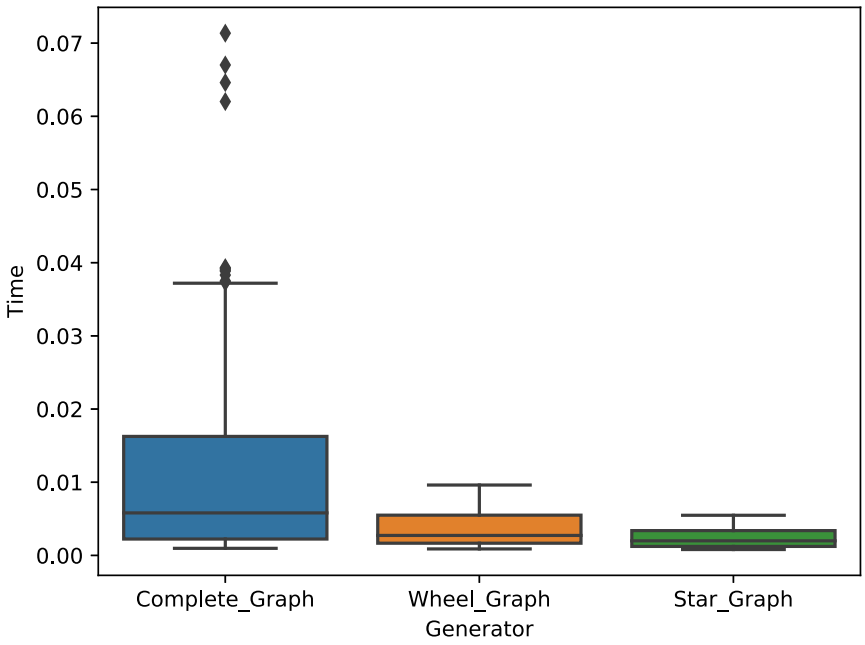


Figure 1: Boxplot of the graph generators

In this graph differences are seen between groups, especially in Complete Graph which has higher values than the other two graph generators. Also the groups have different distribution.

The second boxplot shows the effect of the maximum flow algorithms in the computation time.
Data for Maximum Flow has a mean of 0.006351 and a standard deviation of 0.009479, whereas data for Minimum Cut has a mean of 0.005868 and a standart deviation of 0.009092, and the Preflow Push has a mean of 0.005480 and a standard deviation of 0.007968.

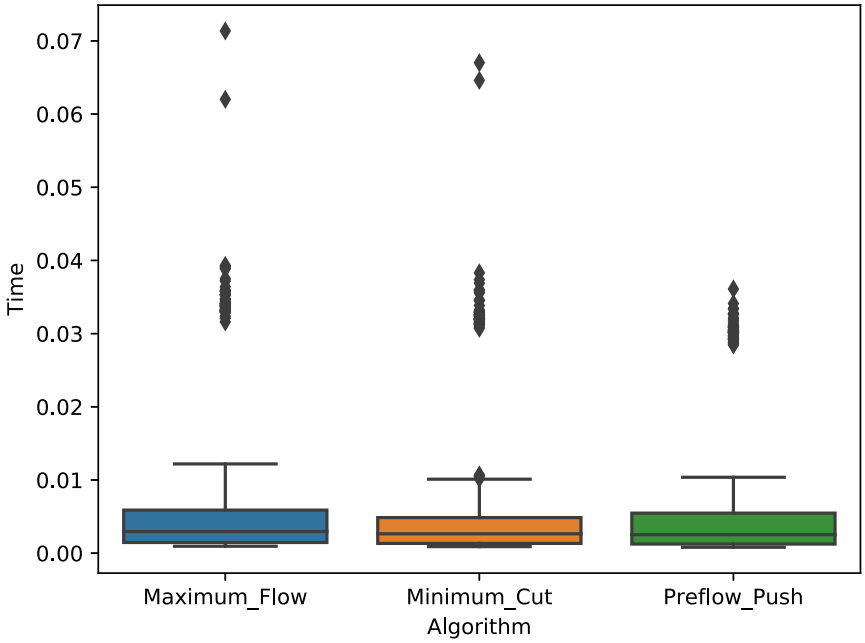


Figure 2: Boxplot of maximum flow algorithms

From the graph can be seen some similarities among the group of algorithms but further analysis must be perform to corroborate this.
The third boxplot shows the effect of the graph order in the computation time.
Data for Order 4 has a mean of 0.001124 and a standard deviation of 0.000144, whereas data for Order 8 has a mean of 0.002206 and a standart deviation of 0.000651, Order 16 has a mean of 0.005231 and a standard deviation of 0.002970 and Order 32 has a mean of 0.015039 and a standard deviation of 0.013616.

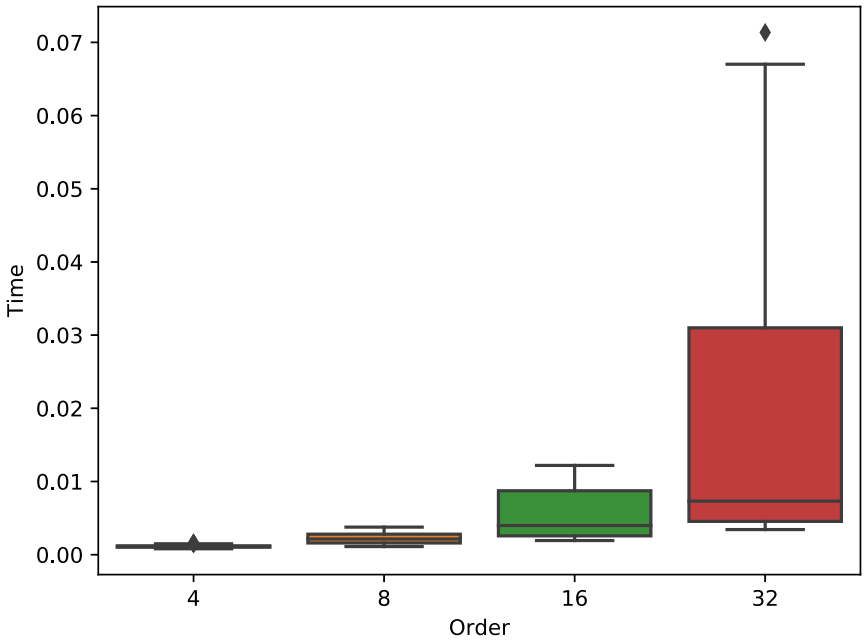


Figure 3: Boxplot of graph order

Here a difference in order is clearly seen. As the order grows the execution time grows as well, so it can be think they may be positive correlated.
The fourth boxplot shows the effect of the eight values of the density in the computation time. Data of the mean and standart deviation is shown below.

Table 1: Mean and Standart Deviation values for graph densities

	0.06	**0.12**	**0.13**	**0.25**	**0.29**	**0.50**	**0-67**	**1.00**
Mean	0.004190	0.007376	0.002408	0.004039	0.001476	0.002168	0.001014	0.009625
Standard Deviation	0.000507	0.000829	0.000300	0.000540	0.000189	0.000209	0.000112	0.012618

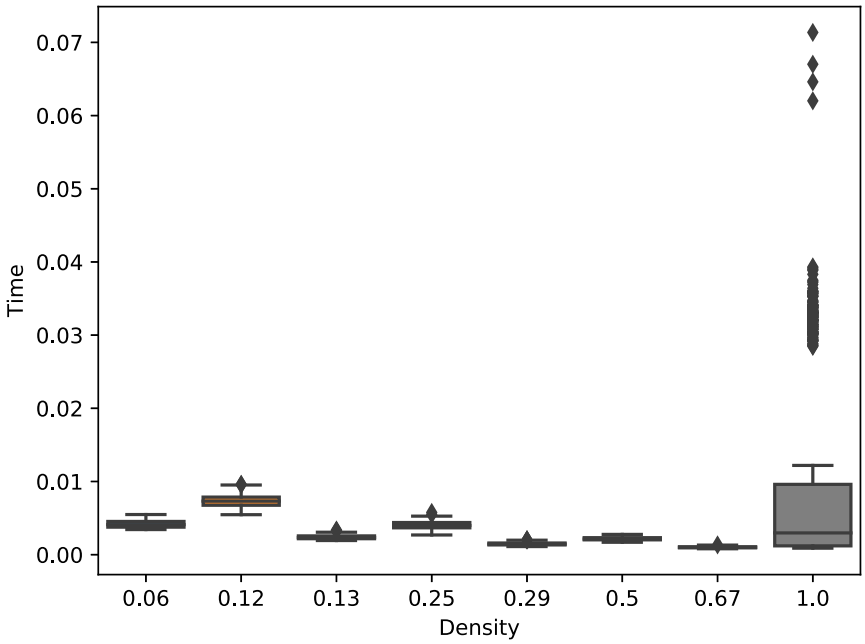


Figure 4: Boxplot of density

Here it can be seen the effect of the density in the execution time, showing differences in the distribution of each group.

In order to see if these effects have interactions an ANOVA is performed.

ANOVA.txt

	sum_sq	df	F	PR>F
Generator	5.250796e-04	2.0	56.267109	2.015921e-24
Algorithm	2.280564e-04	2.0	24.438344	3.385212e-11
Generator:Algorithm	1.227626e-05	4.0	0.657757	6.214187e-01
Order	7.006502e-02	1.0	15016.222797	0.000000e+00
Generator:Order	1.022268e-04	2.0	10.954543	1.868646e-05
Order:Algorithm	2.821883e-04	2.0	30.239076	1.217277e-13
Density	2.540693e-03	1.0	544.517337	2.639616e-105
Generator:Density	4.396328e-05	2.0	4.711070	9.107547e-03
Algorithm:Density	8.825691e-05	2.0	9.457540	8.209189e-05
Order:Density	2.257209e-07	1.0	0.048376	8.259386e-01
Residual	8.310065e-03	1781.0	NaN	NaN
C(Fancy)Verbatim	8.310065e-03	1781.0	NaN	NaN

From the ANOVA it can be said p-values are quite small, so there are differences among group means, concluding that all variables are related with the execution time

For an analysis of correlation it is performed a scatter matrix, showing the relation of the quantitative variables and its corresponding scatterplots, the kernel density estimation as a way to estimate the probability density function of the variables and the coeficients of correlation between the variables, where it can be seen the highest correlation is between the variables time and order, with a value of 0.613, but still it is not a strong relation.

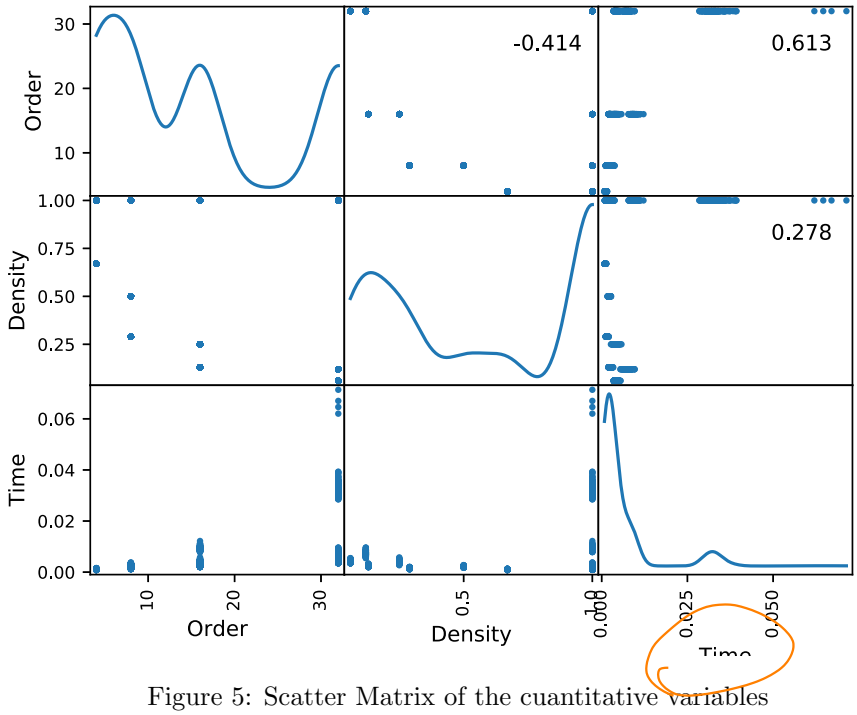


Figure 5: Scatter Matrix of the cuantitative variables

References

[1] Paul Erdös. Graph theory and probability. *Canadian Journal of Mathematics*, 11:34–38, 1959.

[2] Giorgio Gallo, Michael D Grigoriadis, and Robert E Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.

[3] Valerie King, Satish Rao, and Rorbert Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[4] Victor E. Mendia and Dilip Sarkar. Optimal broadcasting on the star graph. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):389–396, 1992.

[5] NetworkX. Source code for networkx.drawing.layout, 2018. `https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html`, Last accessed on 2019-03-31.

[6] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4): 585–591, 1997.

Homework Assignment 4: Network Flows

5273

Introduction

For this work graph generators of the library NetworkX of Python have been used. Three of these generators: Complete Graph, Wheel Graph, and Star Graph have been selected for the study. To use each one of them a base two as logarithmic base to generate four different graph orders is chosen, so the total number of nodes 4, 8, 16 and 32 nodes. The total number of edges depends on the chosen graph. Edge weights have been generated following a normally distribution with mean of 12 units and a standard deviation of 0.4.
After the assigment of normally distributed weights to edges, these weights would be used as instances of the maximum flow problem. Three implementations of this problem are executed using NetworkX algorithms: Maximum Flow, Minimum Cut and Preflow Push. Each one implementations are performed for each one of the graph generators.
This work it is run on an Intel® Celeron CPU @ 1.10 GHz with 4 GB RAM Laptop.

Graph Generators

Complete Graph

The first graph generator chosen is the complete graph. This is a simple undirected graph which each pair of nodes is connected by a unique edge [1]. The complete graphs selected, as mentioned in Introduction, have 4, 8, 16, and 32 nodes with 6, 20, 120, and 496 edges respectively.

Wheel Graph

The second graph generator chosen is the wheel graph. This is a simple undirected graph characterized by its single hub node connected to each of the $(n-1)$-node cycle graph [5]. The wheel graphs selected, as well as the other graph generators, have 4, 8, 16 and 32 nodes with 6, 14, 30, and 462 edges respectively.

Star Graph

The third graph generator chosen is the star graph. This is a simple undirected graph that it can be considered as a tree with one internal node and k leaves [4]. The star graphs selected, as well as the other graph generators, have an order of 4, 8, 16 and 32 nodes with also 4, 8, 16, and 32 edges respectively.

```
for j in range(10):
    for i in range(2,6):
        i = 2 ** i
        number_of_nodes.append(i)

        #COMPLETE GRAPH
        start_C = time.time()
        C=nx.complete_graph(i)
        end_C = time.time()
        execution_time_C = end_C - start_C
        complete_graph_times.append(execution_time_C)

        assign_normally_distributed_weight(12,0.4,C)

        # WHEEL GRAPH
        start_W = time.time()
        W = nx.wheel_graph(i)
        end_W = time.time()
        execution_time_W = end_W - start_W
        wheel_graph_times.append(execution_time_W)

        assign_normally_distributed_weight(12, 0.4, W)

        # STAR GRAPH
        start_S = time.time()
        S = nx.star_graph(i)

        end_S = time.time()
        execution_time_S = end_S - start_S
        star_graph_times.append(execution_time_S)

        assign_normally_distributed_weight(12, 0.4, S)
```

graphs_t4.py

Maximum Flow Algorithms

Max Flow

The algorithm can be used in the three chosen graph generators since the settings of the arc capacities are not fixed and are functions of a single parameter [2], it have been defined the source and the sink in each graph and a positive capacity for every edge. The value of the flow is the net flow out of the source. The maximum flow problem is that of finding a flow of maximum value [3].

```
            #Maximum flow Complete Graph
            start_flow = time.time()
            for k in range(5):
                flow_value = nx.maximum_flow(C, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Maximum flow Wheel Graph
            start_flow = time.time()
            for k in range(5):
                flow_value = nx.maximum_flow(W, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
# Maximum flow Star Graph
start_flow = time.time()
for k in range(5):
    flow_value = nx.maximum_flow(S, ss_list[k][0], ss_list[k][1], capacity='weight')
end_flow = time.time()
execution_time_flow = end_flow - start_flow
```

graphs_t4.py

Minimum Cut

This algorithm performed well with the three chosen graph generators, which are undirected edge-weighted graphs. The minimum cut of an undirected graph with edge weights consists in a set of edges with minimum sum of weights, such that its removal would cause the graph to become inconnected [6].

```
#Minimum cut Complete Graph
start_flow = time.time()
for k in range(5):
    cut_value = nx.minimum_cut(C, ss_list[k][0], ss_list[k][1], capacity='weight')
end_flow = time.time()
execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
# Minimum cut Wheel Graph
start_flow = time.time()
for k in range(5):
    cut_value = nx.minimum_cut(W, ss_list[k][0], ss_list[k][1], capacity='weight')
end_flow = time.time()
execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
# Minimum cut Star Graph
start_flow = time.time()
for k in range(5):
    cut_value = nx.minimum_cut(S, ss_list[k][0], ss_list[k][1], capacity='weight')
end_flow = time.time()
execution_time_flow = end_flow - start_flow
```

graphs_t4.py

Preflow Push

This implementation finds a maximum single-commodity flow using the highest-label preflow-push algorithm [5] . It returns the residual network resulting after computing the maximum flow.

```
#Preflow push Complete Graph
start_flow = time.time()
for k in range(5):
    R = preflow_push(C, ss_list[k][0], ss_list[k][1], capacity='weight')
end_flow = time.time()
execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
# Preflow push Wheel Graph
start_flow = time.time()
for k in range(5):
    R = preflow_push(W, ss_list[k][0], ss_list[k][1], capacity='weight')
end_flow = time.time()
execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Preflow push Star Graph
            start_flow = time.time()
            for k in range(5):
                R = preflow_push(S, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
            preflow_push_times.append(execution_time_flow)
```

graphs_t4.py

Experimental Results

Once the corresponding graphs have been generated can be evaluated through statistical methods and visualizations the effect of the used graph generator in the computation time, te maximum flow algorithms, the graph order, and the density of the graph.
The first boxplot shows the effect of the graph generator in the computation time.
Data for Complete Graph has a mean of 0.011743807 and a standard deviation of 0.013, whereas data for Wheel Graph has a mean of 0.0022 and a standart deviation of 0.0012, and the Star Graph Generator has a mean of 0.0036 and a standard deviation of 0.0024.

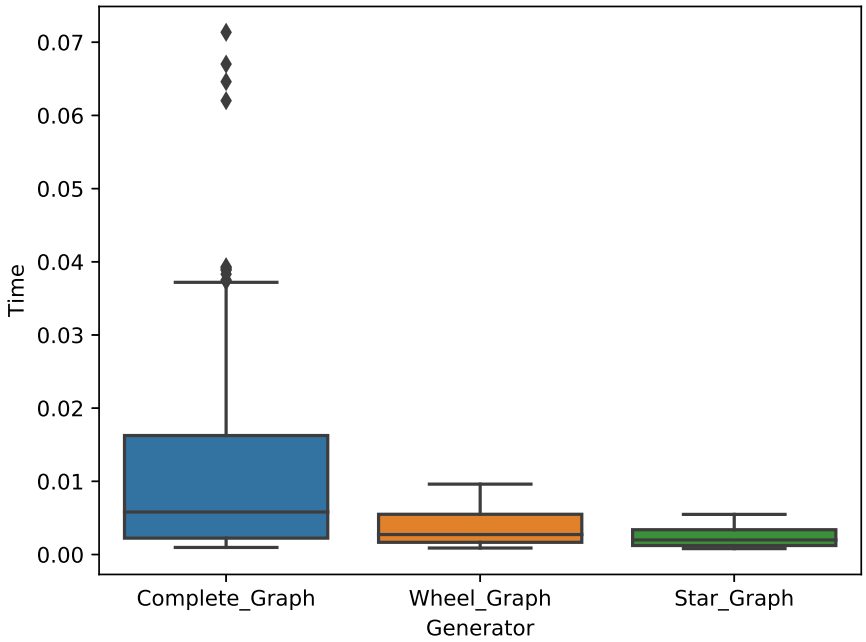


Figure 1: Boxplot of the graph generators

In this graph differences are seen between groups, especially in Complete Graph which has higher values than the other two graph generators. Also the groups have different distribution.

The second boxplot shows the effect of the maximum flow algorithms in the computation time.
Data for Maximum Flow has a mean of 0.006 and a standard deviation of 0.009, whereas data for Minimum Cut has a mean of 0.005 and a standart deviation of 0.009, and the Preflow Push has a mean of 0.005 and a standard deviation of 0.007.

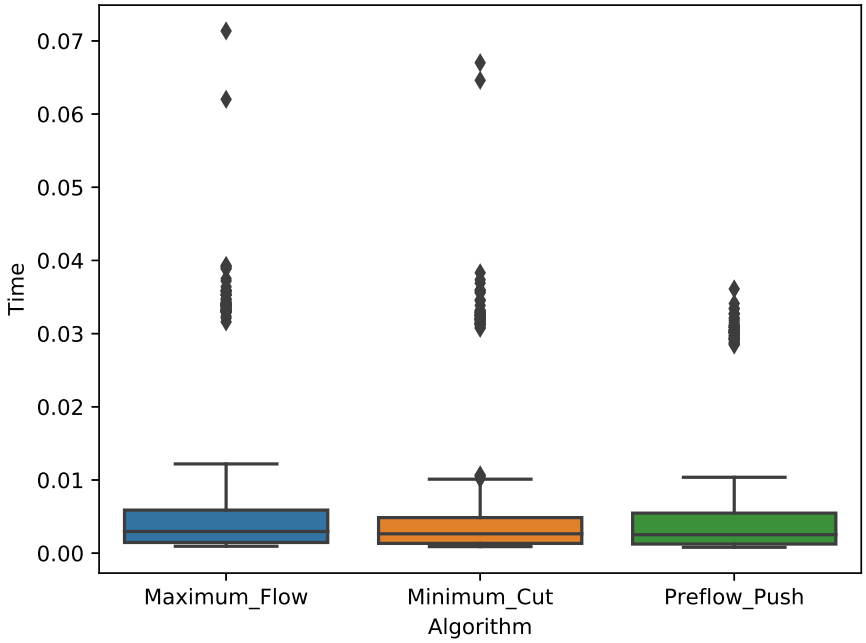


Figure 2: Boxplot of maximum flow algorithms

From the graph can be seen some similarities among the group of algorithms but further analysis must be perform to corroborate this.
The third boxplot shows the effect of the graph order in the computation time.
Data for Order 4 has a mean of 0.001 and a standard deviation of 0.0001, whereas data for Order 8 has a mean of 0.002 and a standart deviation of 0.0006, Order 16 has a mean of 0.005 and a standard deviation of 0.002 and Order 32 has a mean of 0.015 and a standard deviation of 0.013.

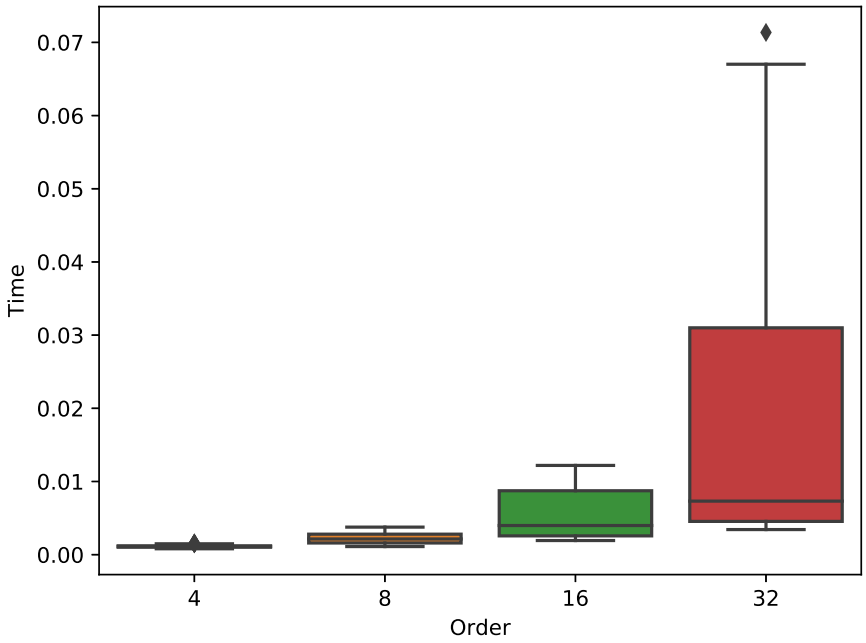


Figure 3: Boxplot of graph order

Here a difference in order is clearly seen. As the order grows the execution time grows as well, so it can be thought they may be positive related.
The fourth boxplot shows the effect of the eight values of the density in the computation time. Data of the mean and standart deviation is shown below.

Density	**0.06**	**0.12**	**0.13**	**0.25**	**0.29**	**0.50**	**0.67**	**1.00**
Mean	0.004190	0.007376	0.002408	0.004039	0.001476	0.002168	0.001014	0.009625
Standard Deviation	0.000507	0.000829	0.000300	0.000540	0.000189	0.000209	0.000112	0.012618

Table 1: Mean and Standard Deviation values for graph densities

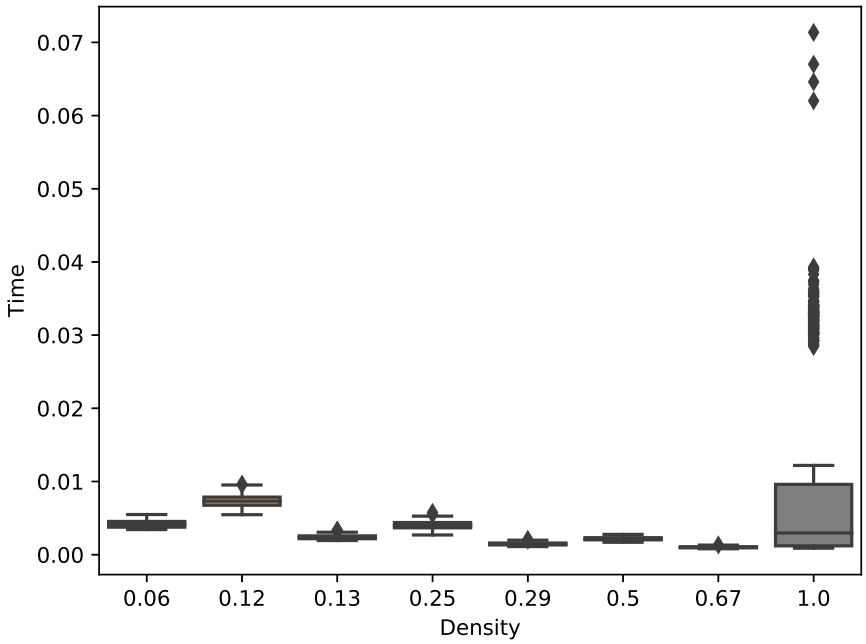


Figure 4: Boxplot of density

Here it can be seen the effect of the density in the execution time, showing differences in the distribution of each group.

In order to see if these effects have interactions an ANOVA is performed.

	sum_sq	df	F	PR(>F)
Generator	5.250796e−04	2	56.26	<0.0001
Algorithm	2.280564e−04	2	24.43	<0.0001
Generator : Algorithm	1.227626e−05	4	0.65	<0.0001
Order	7.006502e−02	1	15016.22	<0.0001
Generator : Order	1.022268e−04	2	10.95	<0.0001
Order : Algorithm	2.821883e−04	2	30.23	<0.0001
Density	2.540693e−03	1	544.51	<0.0001
Generator : Density	4.396328e−05	2	4.71	<0.0001
Algorithm : Density	8.825691e−05	2	9.45	<0.0001
Order : Density	2.257209e−07	1	0.04	<0.0001
Residual	8.310065e−03	1781		

ANOVA.txt

From the ANOVA it can be said p-values are quite small, so there are differences among group means, concluding that all variables are related with the execution time

For an analysis of correlation a scatter matrix is drawn, showing the relation of the quantitative variables and its corresponding scatterplots, the kernel density estimation as a way to estimate the probability density function of the variables and the coeficients of correlation between the variables, where it can be seen the highest correlation is between the variables time and order, with a value of 0.613, but still it is not a strong relation.

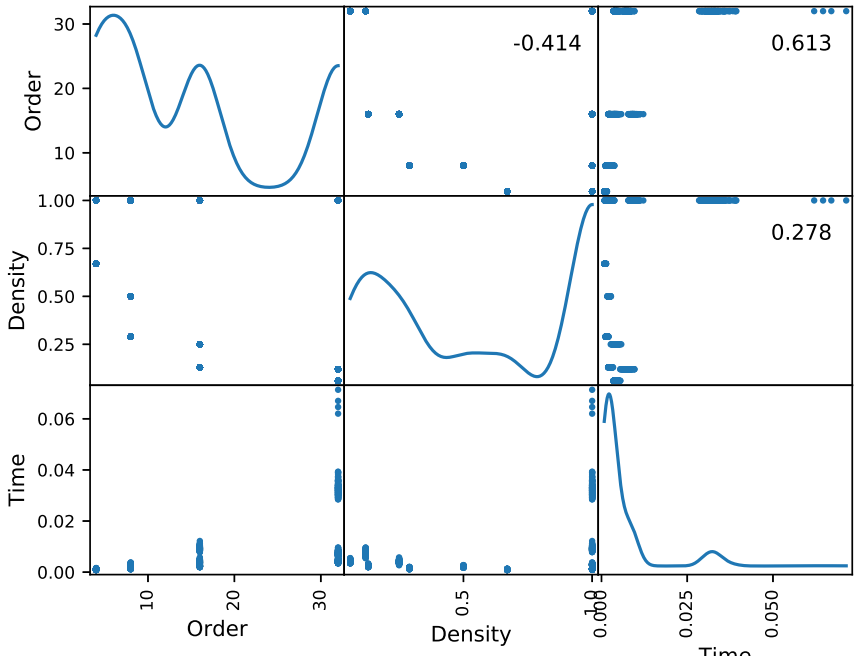


Figure 5: Scatter Matrix of the cuantitative variables

References

[1] Paul Erdös. Graph theory and probability. *Canadian Journal of Mathematics*, 11:34–38, 1959.

[2] Giorgio Gallo, Michael D Grigoriadis, and Robert E Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.

[3] Valerie King, Satish Rao, and Rorbert Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[4] Victor E. Mendia and Dilip Sarkar. Optimal broadcasting on the star graph. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):389–396, 1992.

[5] NetworkX. Source code for networkx.drawing.layout, 2018. `https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html`, Last accessed on 2019-03-31.

[6] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591, 1997.

Homework 5 Corrections

In this homework corrections in notations were made. The selected misused words and phrases in the feedback were changed and also punctuation marks were corrected.

Homework Assignment 5: Network Flows

5273

Introduction

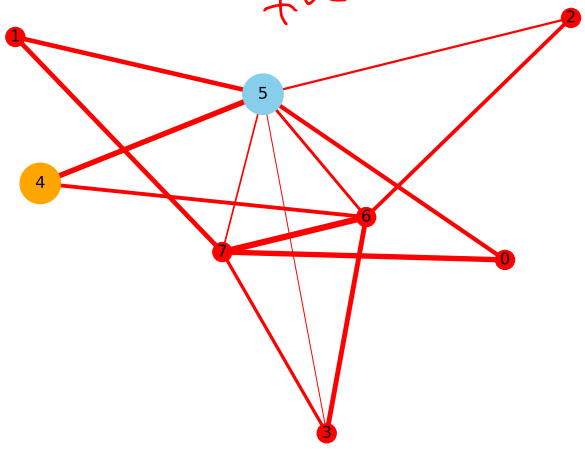
For this work the library NetworkX of Python have been use to generate an undirected powerlaw cluster graph of 8 nodes [4]. This graph it is used to define the instances, characterized by the graph and different pairs of source and targets. Edge weights have been generated following a normal distribution with mean of 3 units and a standard deviation of 1.5. This data is is used to generate a system where a maximum flow problem needs to be solved.
This work is run on an Intel Celeron CPU @ 1.10 GHz with 4 GB RAM laptop.
The powerlaw cluster graph has been chosen because it perfectly can modelate a logistic supply chain network where every node represents each logistic point and the edges the possible connections between every point[3]. The objective is to maximize the flow of value in the network[1].
In order to draw this situation a Fruchterman-Reingold layout is chosen, which seems to be the good way to understand the interactions among nodes because of the attempts to minimize the number of overlapping nodes and edges [5]. Also this layout allows to to distribute vertices evenly, make edge lengths uniform, and reflect symmetry [2].
The nodes are represented by different colors. The skyblue color represents the source nodes whereas the orange color represents the targets nodes. The rest of the nodes are represented in red. Another visual modification made is that sources and target nodes are represented with a bigger size than the other nodes.
In the case of edges they are represented in black color. The width of the edge indicates its capacity, for example, wider edges are the ones with greater capacity. When the maximum flow algorithm is executed those edges where the flow passes are colored in red in order to indicate the maximum flow produced by the algorithm.

```python
for k in range(5):
    #Maximum flow algorithm:
    flow_value, flow_dict = nx.maximum_flow(G, st_list[k][0], st_list[k][1], capacity='weight')
    #print(flow_dict)

    #Change edge colors
    edge_colors = ['black' if flow_dict[i][j] == 0 and flow_dict[j][i] == 0 else 'red' for i,
    j in G.edges()]

    #Change node properties:
    color_map = []
    node_sizes = []
    for node in G:
        if node == st_list[k][0]:
            color_map.append('skyblue')
            node_sizes.append(900)

        elif node == st_list[k][1]:
            color_map.append('orange')
            node_sizes.append(900)
        else:
            color_map.append('red')
            node_sizes.append(200)

    #Layout:
        pos = nx.fruchterman_reingold_layout(G, k=0.2, iterations=40)

    nx.draw(G, node_color=color_map, edge_color=edge_colors,
            node_size=node_sizes, width=weights, pos=pos,
            with_labels=True)
    plt.show()
```

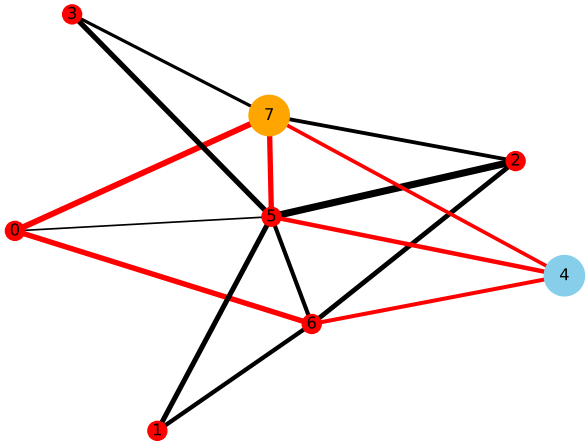
graphs_t5.py

Structural Characterization of Instances

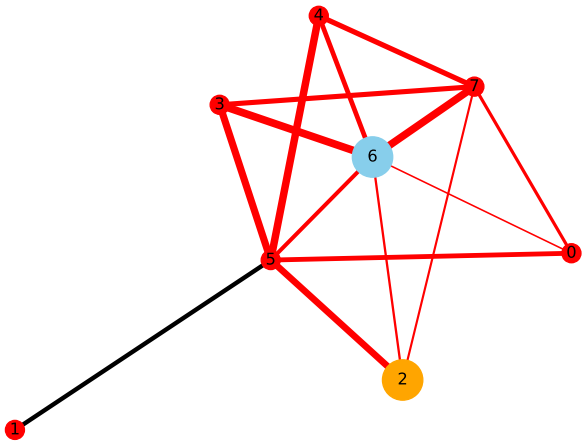
Instances are characterized by a Graph, in this case the powerlaw cluster graph, by a source node and by a target node. For this work 5 instances are generated. The source and target nodes are selected at random.



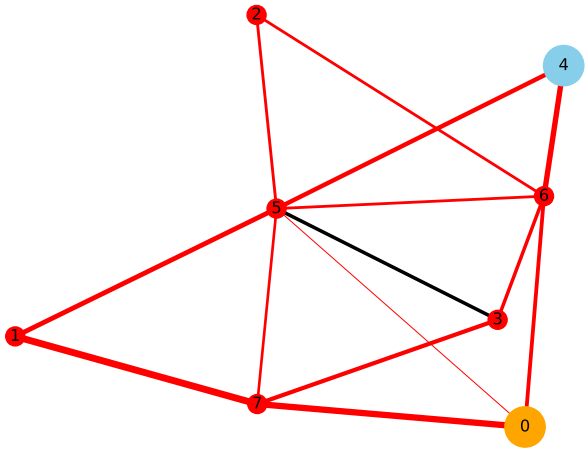
(a) Visualization of Instance 01



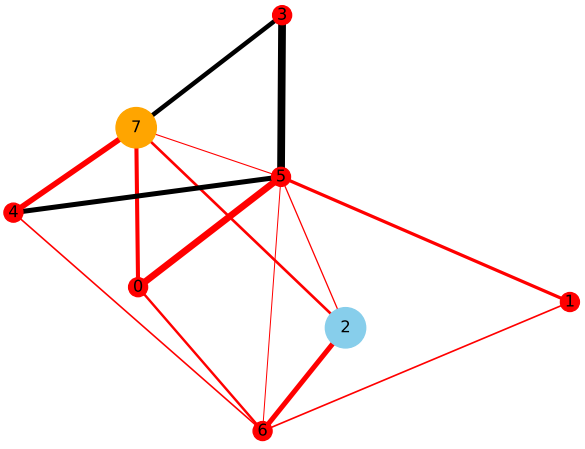
(b) Visualization of Instance 02



(c) Visualization of Instance 03



(d) Visualization of Instance 04



(e) Visualization of Instance 05

Figure 1: Instances

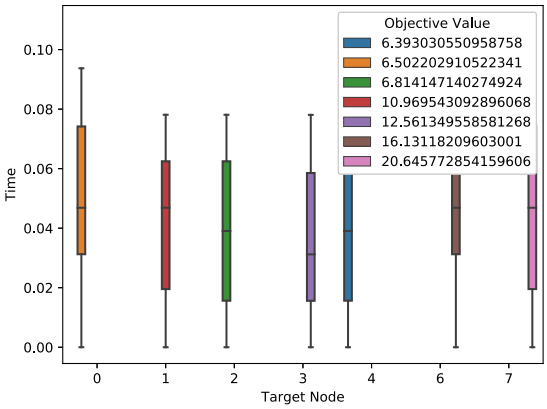
This way the instances are:

- Instance 01: G, 5, 4.
- Instance 02: G, 4, 7.
- Instance 03: G, 6, 2.
- Instance 04: G, 4, 0.
- Instance 05: G, 2, 7.

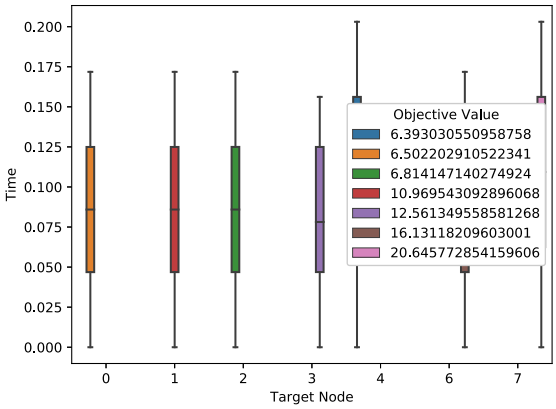
For the visualized graphs the following 6 structural characteristics have been calculated:

- Degree distribution
- Clustering coefficient
- Closeness centrality
- Load centrality
- Eccentricity
- Page Rank

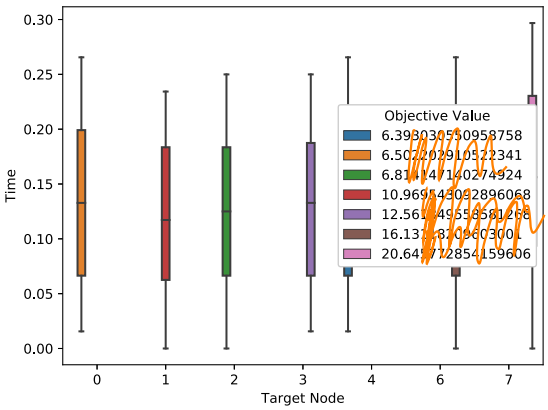
In order to analyze the influence of this characteristics in the execution time as well as in the optimum value boxplots are constructed.
Also an analysis of best sources and targets it is carried out based on the values of the maximum flow algorithm. Three nodes are given to conclude those best values.
For this analysis, within each instance a source node is fixed and the other nodes are variable. Then the maximum flow algorithm is executed and the best three values will be considered as the better ones for that instance. The same procedure is followed for target nodes.
As a result the best target nodes for Instance 01 are the nodes 6, 7 and 1, whereas the best source nodes are 0, 1 and 2 if we want to maximize the flow of the network.
For Instance 02 the best targets are 0, 1 and 3, whereas nodes 6, 5 and 0 are the best sources
In case of Instance 03 nodes 5, 7 and 3 are the best targets and nodes 3, 4 and 5 are the best sources.
For Instance 04 the results are 3, 5 and 7 as best targets and 2, 3 and 4 as the best sources.
Last, for Instance 05 the best targets are 0, 1 and 3 whereas the best sources are 5, 6 and 0.
Moreover, it is important to investigate the effect of the structural characteristics of vertices in the computation time. In the case of Instance 01 Computation Time seems to be quite similar but with a little different in case of node 7 which have the greater objective value and the longest time. These results are shown in the figures below:



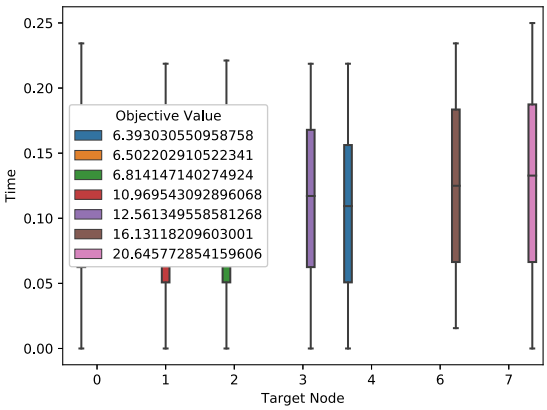
(a) Boxplot of Degree Distribution



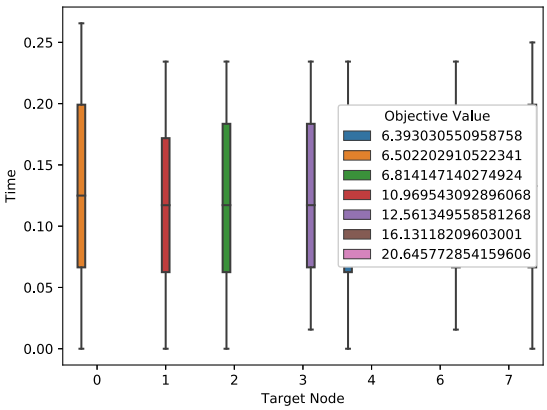
(b) Boxplot of Clustering Components



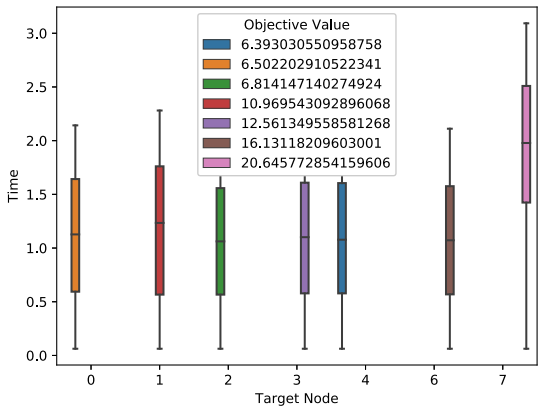
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



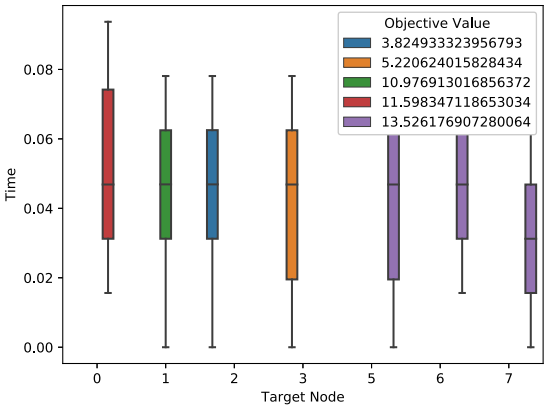
(e) Boxplot of Eccentricity



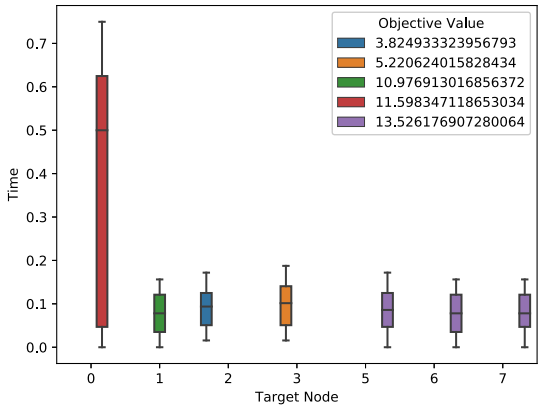
(f) Boxplot of Page Rank

Figure 2: Boxplots of Instance 01

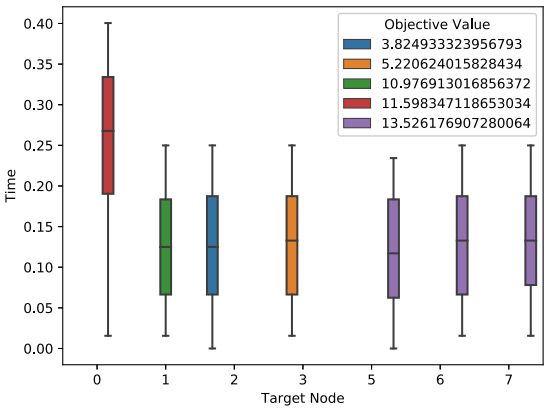
For Instance 02 we have something similar like Instance 01, where time is different in case of some node 0, being the longest in the characteristics Degree Distribution, Clustering Componets and Closeness Centrality, the rest is practically the same. Also this node has the second highest flow value. Here the maximum flow values are lower than Instance 01.



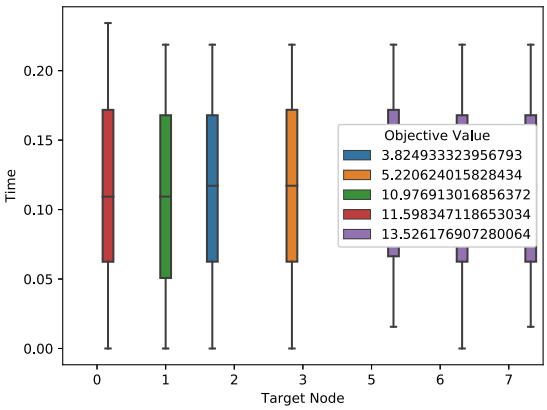
(a) Boxplot of Degree Distribution



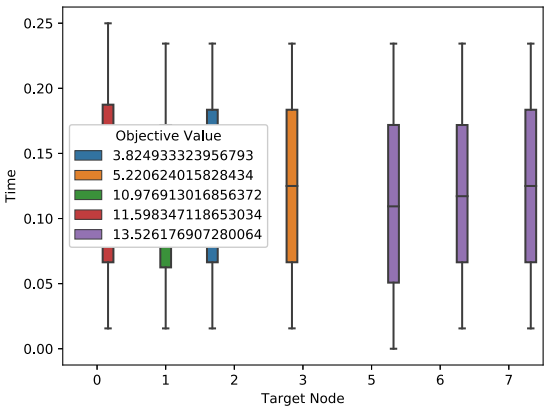
(b) Boxplot of Clustering COmponents



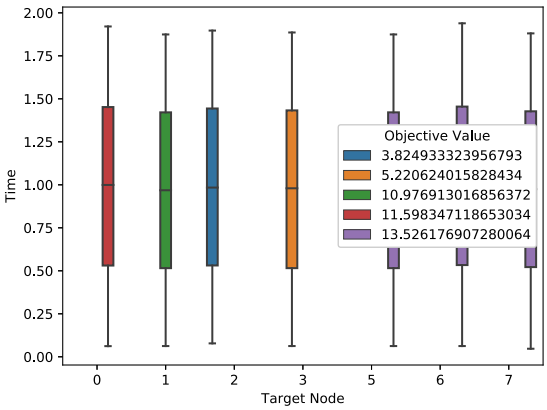
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



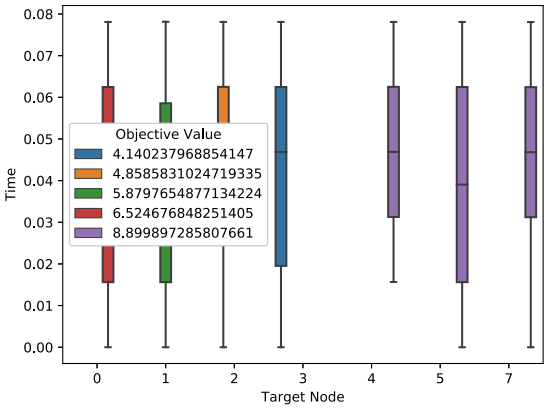
(e) Boxplot of Eccentricity



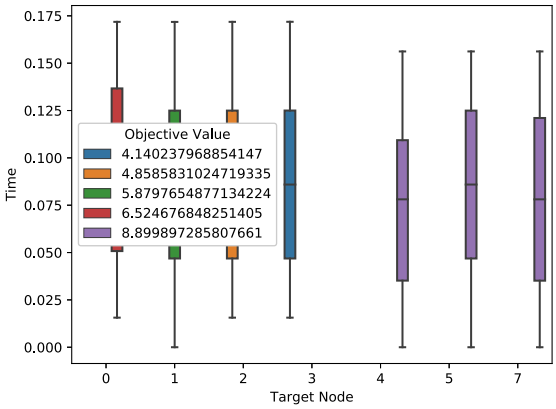
(f) Boxplot of Page Rank

Figure 3: Boxplots of Instance 02

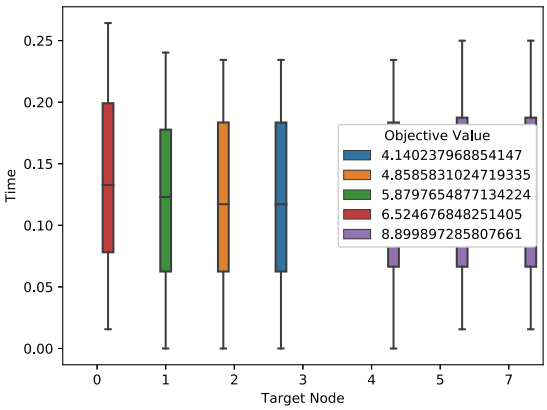
For Instance 03 the time variable is almost the same in all cases, it could be due to the closest values of maximum flow, and also they are not high in comparison with the other instances.



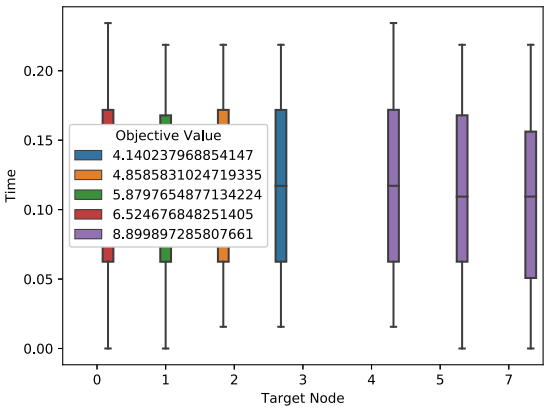
(a) Boxplot of Degree Distribution



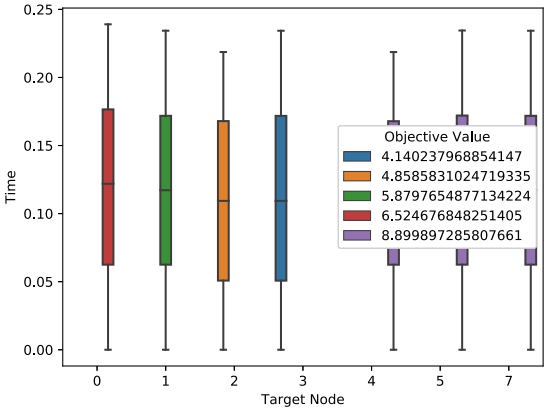
(b) Boxplot of Clustering COmponents



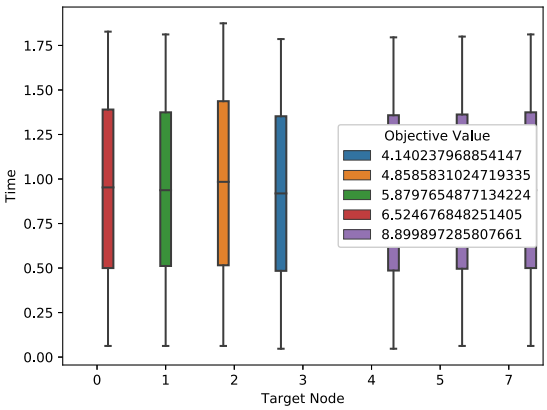
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



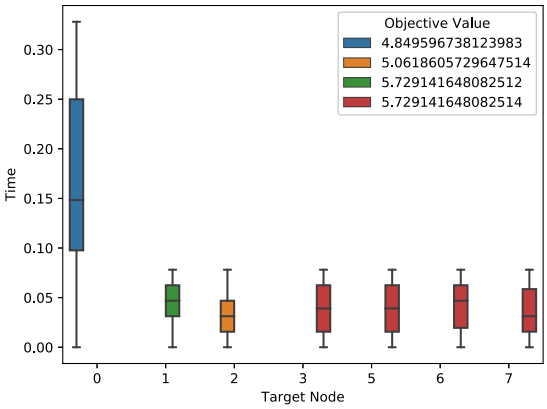
(e) Boxplot of Eccentricity



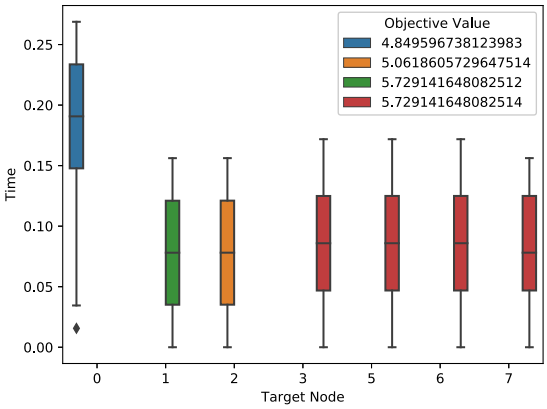
(f) Boxplot of Page Rank

Figure 4: Boxplots of Instance 03

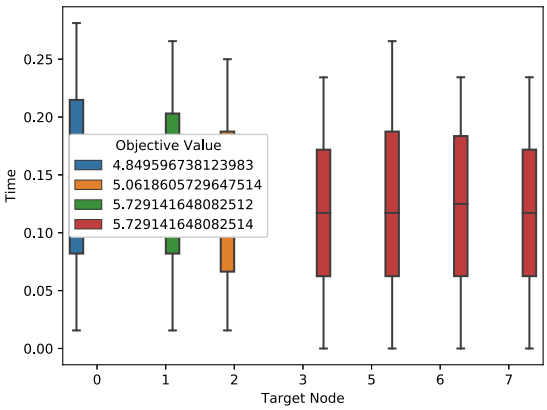
In the case of Instance 04 it can be found the lowest values of flow for the instances tested. Here in some cases node 0 has some peacks in execution time in the case of Degree Distribution and Clustering Components. Also node 5 presents a higher time for Load Centrality and Eccentricity.



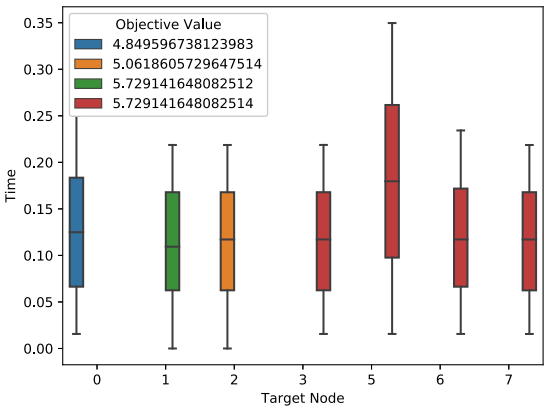
(a) Boxplot of Degree Distribution



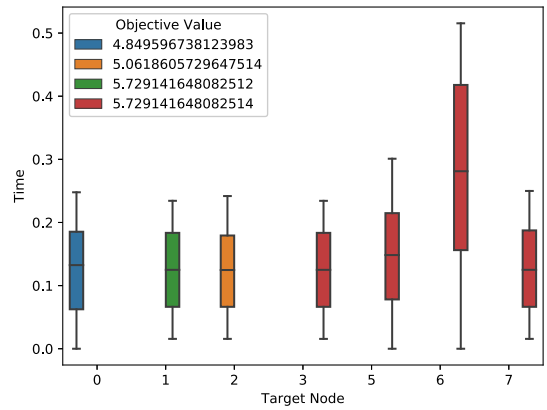
(b) Boxplot of Clustering COmponents



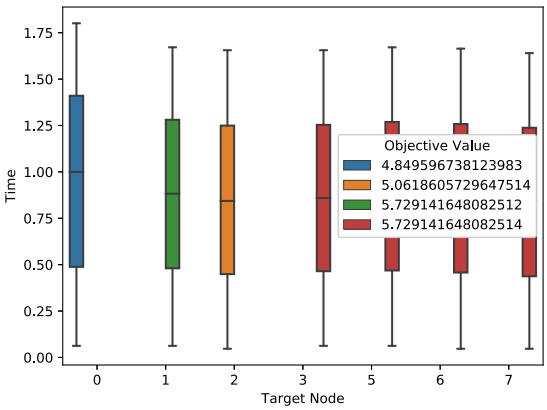
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



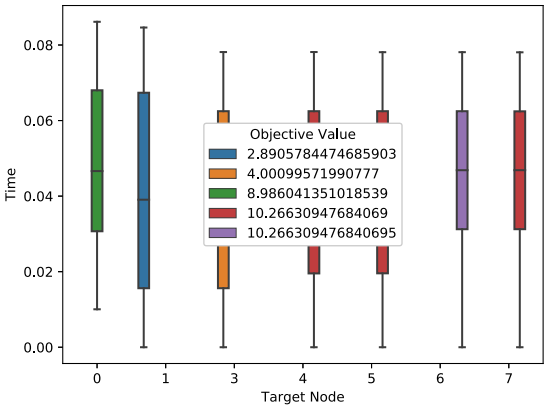
(e) Boxplot of Eccentricity



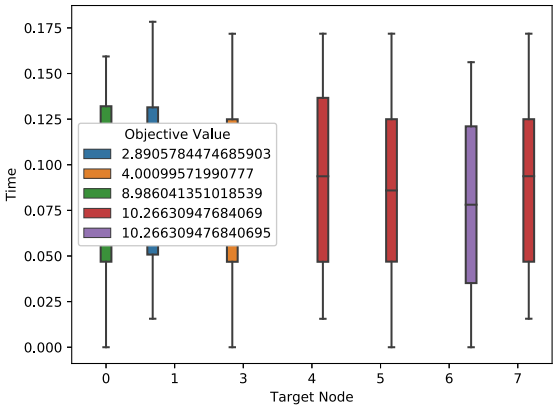
(f) Boxplot of Page Rank

Figure 5: Boxplots of Instance 04

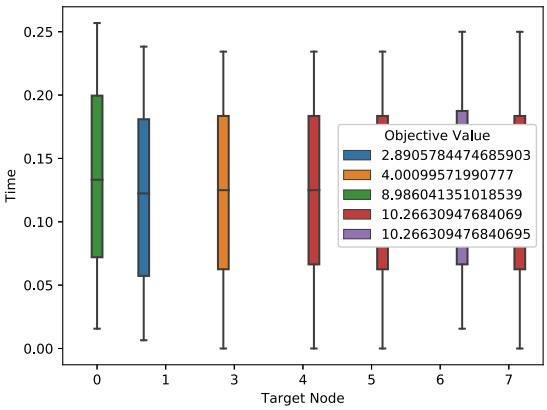
For Instance 05 all data seems to be similar without any node standing out from others as the figure shows below.



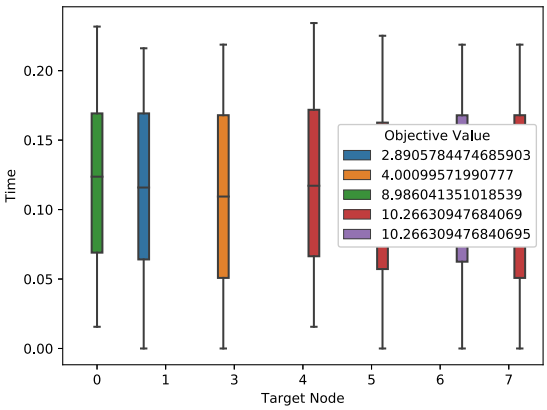
(a) Boxplot of Degree Distribution



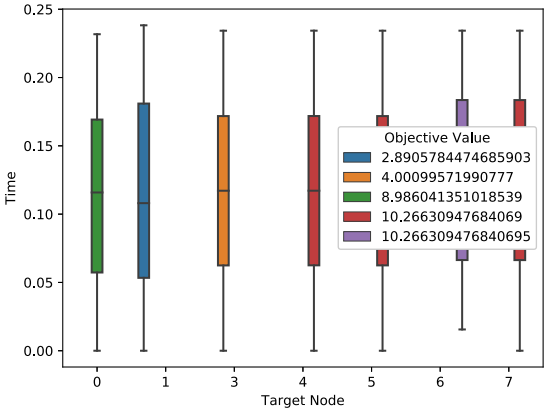
(b) Boxplot of Clustering COmponents



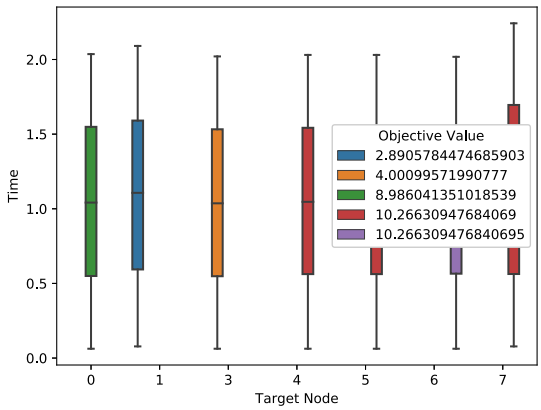
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



(e) Boxplot of Eccentricity



(f) Boxplot of Page Rank

Figure 6: Boxplots of Instance 05

As a conclution it can be said that the maximum flow value obtained is 20.65 units with Instance 01 being node 7 the target and node 5 the source. On the other hand Instance 03 is the one with lowest objectives values with nodes 2 and six Indicating they are not good target and source nodes for maximizing flow.
Page Rank and Closeness centrality are the slowest characteristics in terms of execution time, whereas the fastest characteristics are Degree distribution and Clustering Components but we can not say there is a correlation between the objective value and the execution time. So, aumenting the number of instances results more accurated could be reached.

References

[1] Paul Erdös. Graph theory and probability. *Canadian Journal of Mathematics*, 11:34–38, 1959.

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

[3] Giorgio Gallo, Michael D Grigoriadis, and Robert E Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.

[4] NetworkX. Source code for networkx.drawing.layout, 2018. `https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html`, Last accessed on 2019-03-31.

[5] 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.

Homework Assignment 5: Network Flows

5273

Introduction

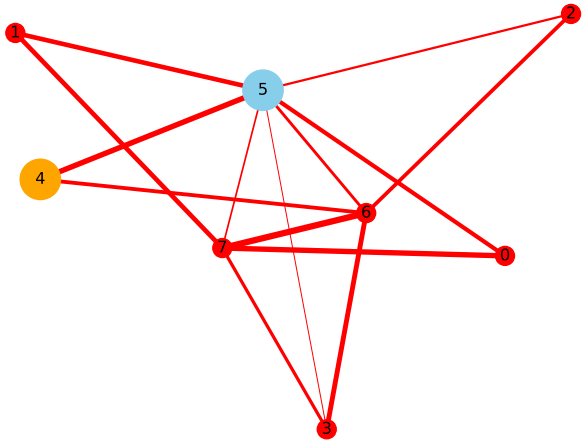
For this work the library NetworkX of Python has been used to generate an undirected powerlaw cluster graph of eight nodes [4]. This graph it is used to define the instances, characterized by the graph and different pairs of source and targets. Edge weights have been generated following a normal distribution with mean of 3 units and a standard deviation of 1.5. This data is used to generate a system where a maximum flow problem needs to be solved.
This work is run on an Intel Celeron CPU @ 1.10 GHz with 4 GB RAM laptop.
The powerlaw cluster graph has been chosen because it can model perfectly a logistic supply chain network where every node represents each logistic point and the edges the possible connections between every point [3]. The objective is to maximize the flow of value in the network [1].
In order to draw this situation a Fruchterman-Reingold layout is chosen, which seems to be a good way to understand the interactions among nodes because of the attempts to minimize the number of overlapping nodes and edges [5]. This layout also allows to distribute vertices evenly, make edge lengths uniform, and reflect symmetry [2].
The nodes are represented by different colors. The skyblue color represents the source nodes whereas the orange color represents the target nodes. The rest of the nodes are represented in red. Another visual modification made is that source and target nodes are represented with a bigger size than the other nodes.
Edges are represented in black. The width of each edge indicates its capacity; for example, wider edges are the ones with greater capacity. When the maximum flow algorithm is executed those edges where the flow passes are colored in red in order to indicate the maximum flow produced by the algorithm.

```
for k in range(5):
    #Maximum flow algorithm:
    flow_value,flow_dict = nx.maximum_flow(G, st_list[k][0], st_list[k][1], capacity='weight')
    #print(flow_dict)

    #Change edge colors
    edge_colors = ['black' if flow_dict[i][j] == 0 and flow_dict[j][i] == 0 else 'red' for i,
    j in G.edges()]

    #Change node properties:
    color_map = []
    node_sizes = []
    for node in G:
        if node == st_list[k][0]:
            color_map.append('skyblue')
            node_sizes.append(900)

        elif node == st_list[k][1]:
            color_map.append('orange')
            node_sizes.append(900)
        else:
            color_map.append('red')
            node_sizes.append(200)

    #Layout:
        pos = nx.fruchterman_reingold_layout(G, k=0.2, iterations=40)

    nx.draw(G, node_color=color_map, edge_color=edge_colors,
            node_size=node_sizes, width=weights, pos=pos,
            with_labels=True)
    plt.show()
```

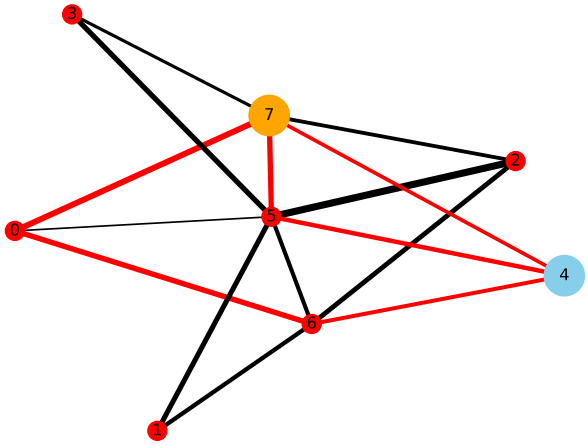
graphs_t5.py

Structural Characterization of Instances

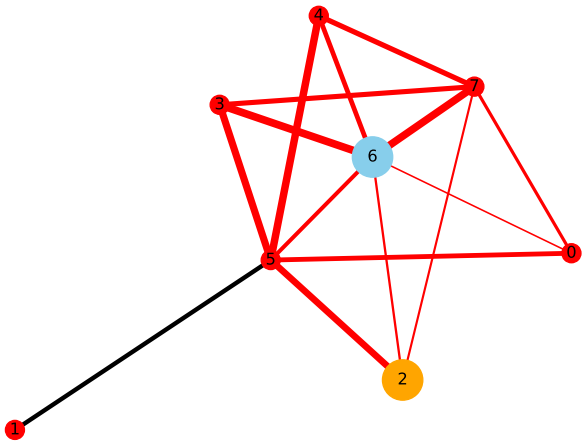
Instances are characterized by a graph, in this case the powerlaw cluster graph, by a source node and by a target node. For this work five instances are generated. The source and target nodes are selected at random.



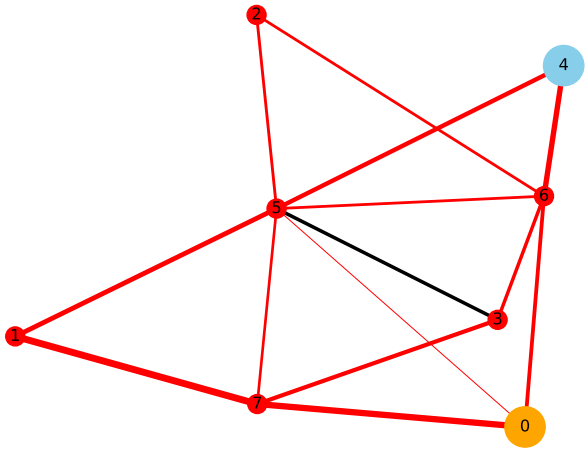
(a) Visualization of Instance 01



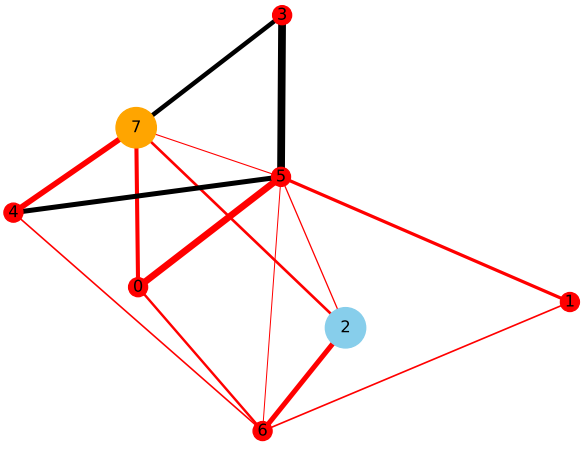
(b) Visualization of Instance 02



(c) Visualization of Instance 03



(d) Visualization of Instance 04



(e) Visualization of Instance 05

Figure 1: Instances

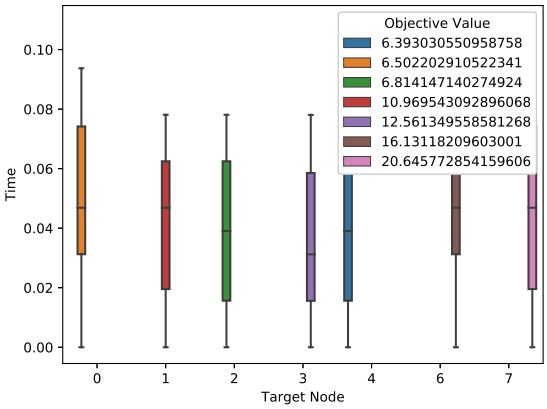
This way the instances are:

- Instance 01: G, 5, 4.
- Instance 02: G, 4, 7.
- Instance 03: G, 6, 2.
- Instance 04: G, 4, 0.
- Instance 05: G, 2, 7.

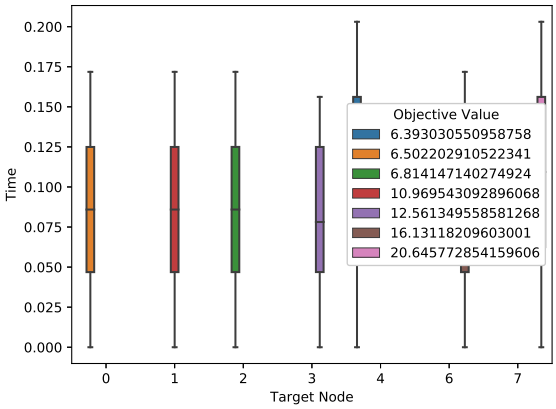
For the visualized graphs the following six structural characteristics have been calculated:

- Degree distribution
- Clustering coefficient
- Closeness centrality
- Load centrality
- Eccentricity
- Page Rank

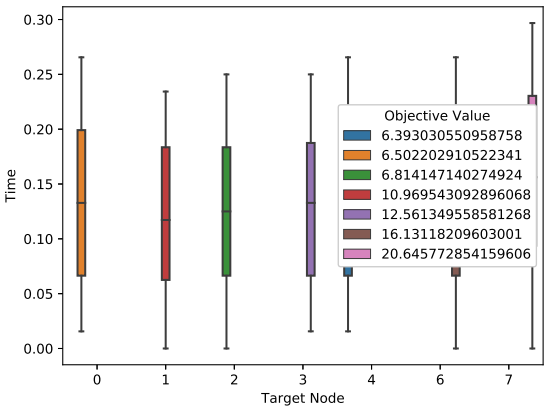
In order to analyze the influence of this characteristics in the execution time as well as in the optimum value boxplots are constructed.
Analysis of best sources and targets it is carried out based on the values of the maximum flow algorithm. Three nodes are given to conclude those best values.
For this analysis, within each instance a source node is fixed and the other nodes are variable. Then the maximum flow algorithm is executed and the best three values are considered as the better ones for that instance. The same procedure is followed for target nodes.
As a result the best target nodes for Instance 01 are the nodes 6, 7 and 1, whereas the best source nodes are 0, 1 and 2 if we want to maximize the flow of the network.
For Instance 02 the best targets are 0, 1 and 3, whereas nodes 6, 5 and 0 are the best sources
In case of Instance 03 nodes 5, 7 and 3 are the best targets and nodes 3, 4 and 5 are the best sources.
For Instance 04 the results are 3, 5 and 7 as best targets and 2, 3 and 4 as the best sources.
Last, for Instance 05 the best targets are 0, 1 and 3 whereas the best sources are 5, 6 and 0.
Moreover, it is important to investigate the effect of the structural characteristics of vertices in the computation time. In the case of Instance 01 Computation Time seems to be quite similar but with a little different in case of node 7 which have the greater objective value and the longest time. These results are shown in the figures:



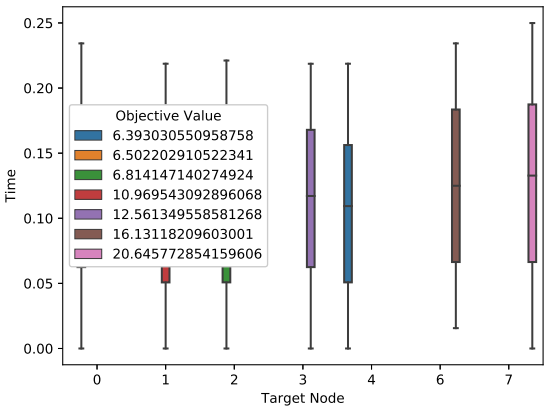
(a) Boxplot of Degree Distribution



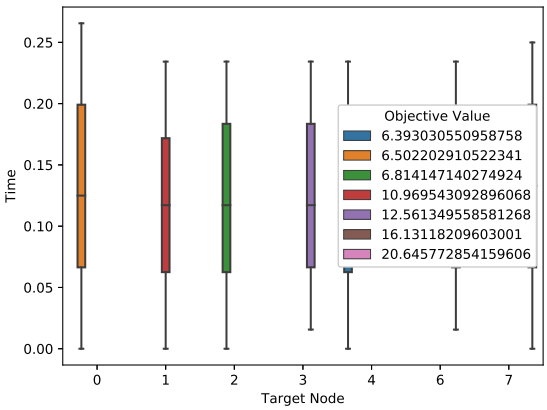
(b) Boxplot of Clustering Components



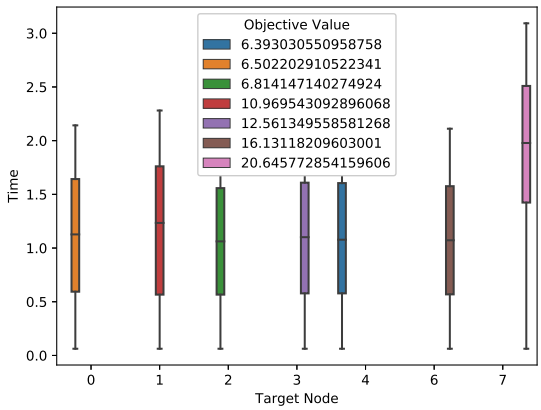
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



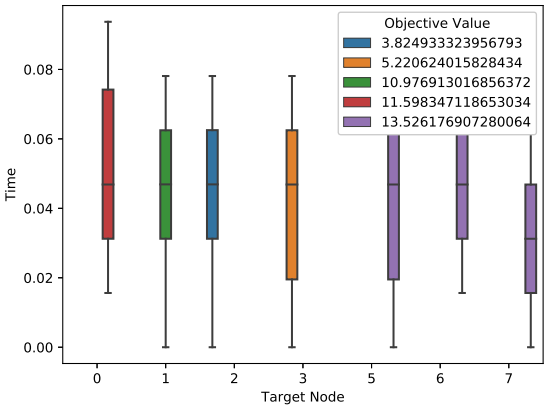
(e) Boxplot of Eccentricity



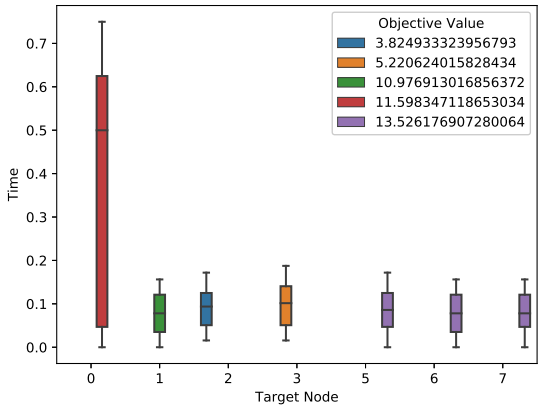
(f) Boxplot of Page Rank

Figure 2: Boxplots of Instance 01

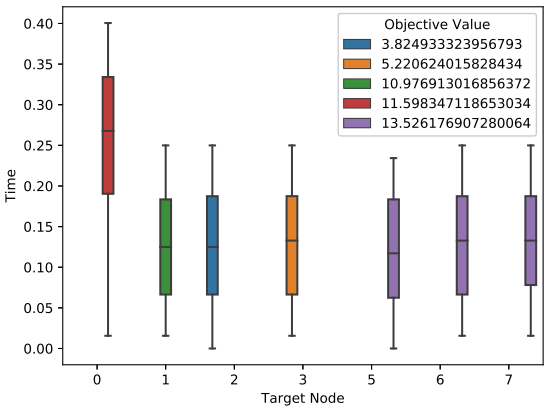
For Instance 02 we have something similar like Instance 01, where time is different in case of some node 0, being the longest in the characteristics Degree Distribution, Clustering Componets and Closeness Centrality, the rest is practically the same. Also this node has the second highest flow value. Here the maximum flow values are lower than Instance 01.



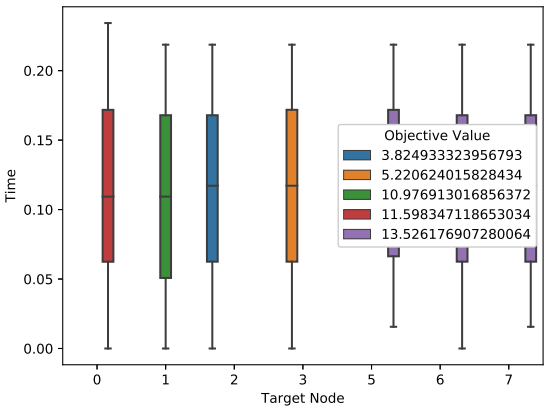
(a) Boxplot of Degree Distribution



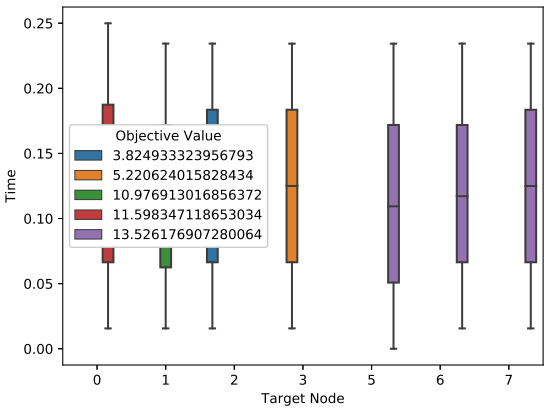
(b) Boxplot of Clustering COmponents



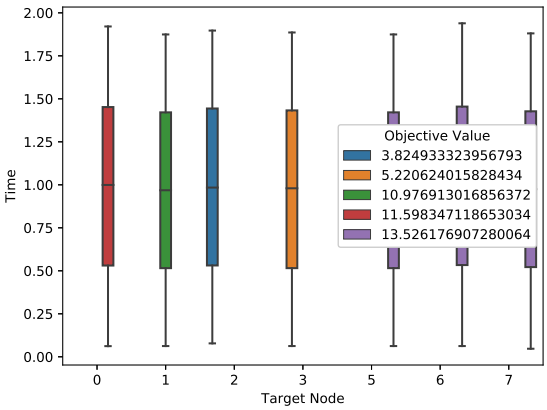
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



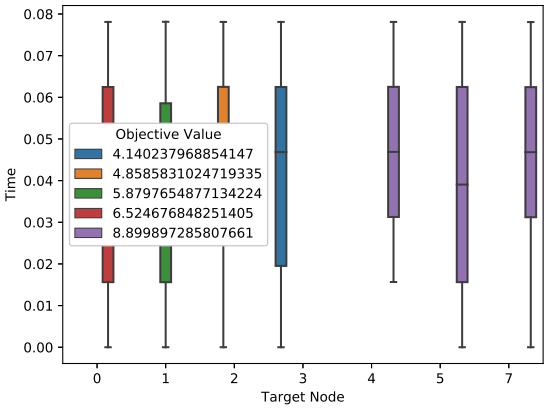
(e) Boxplot of Eccentricity



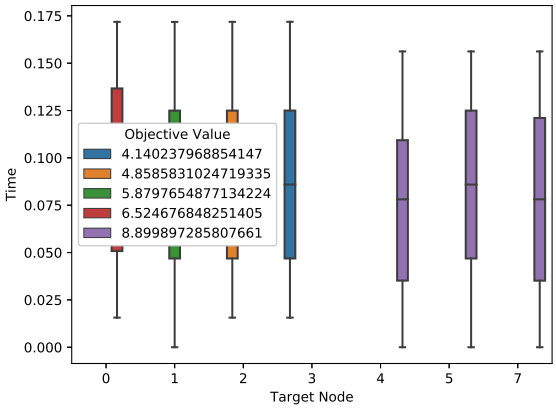
(f) Boxplot of Page Rank

Figure 3: Boxplots of Instance 02

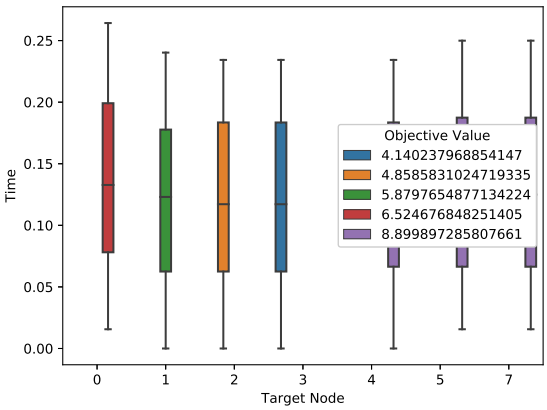
For Instance 03 the time variable is almost the same in all cases, it could be due to the closest values of maximum flow, and also they are not high in comparison with the other instances.



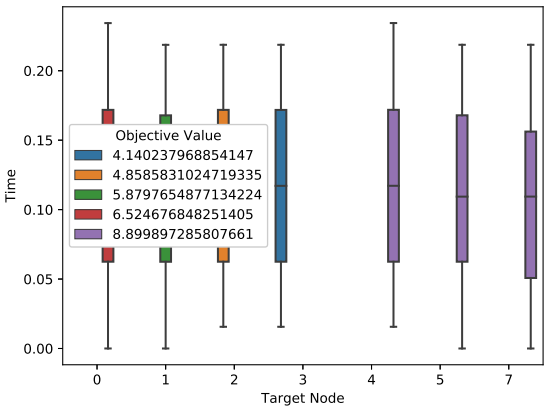
(a) Boxplot of Degree Distribution



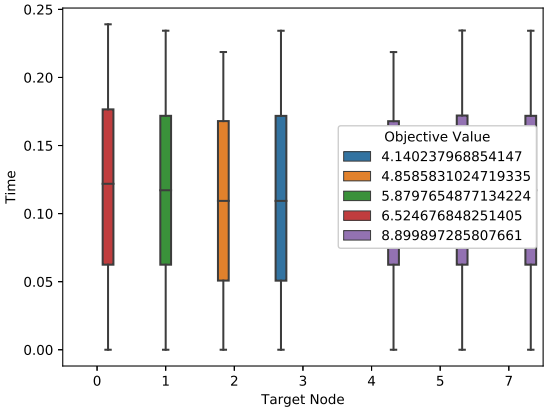
(b) Boxplot of Clustering COmponents



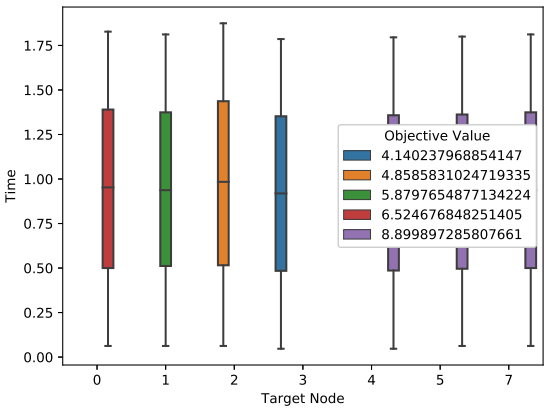
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



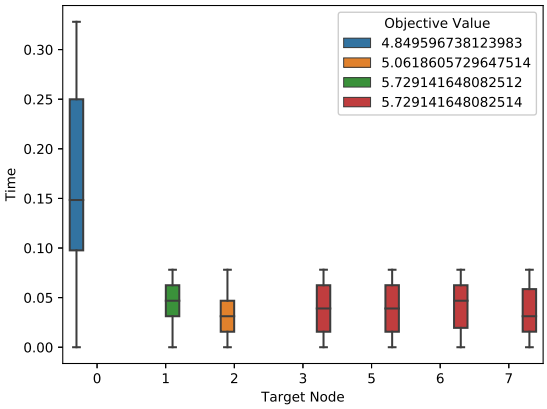
(e) Boxplot of Eccentricity



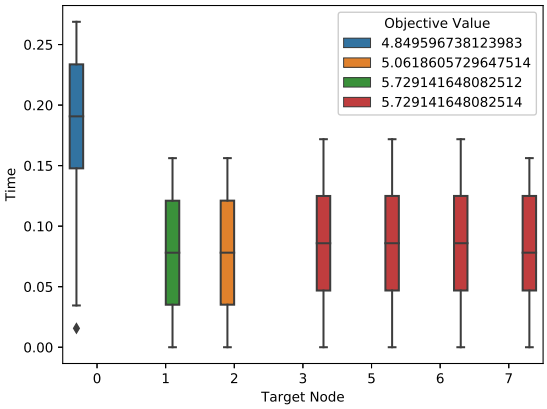
(f) Boxplot of Page Rank

Figure 4: Boxplots of Instance 03

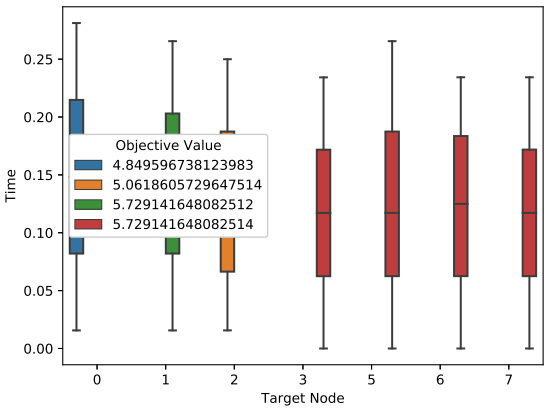
In the case of Instance 04 it can be found the lowest values of flow for the instances tested. Here in some cases node 0 has some peacks in execution time in the case of Degree Distribution and Clustering Components. Also node 5 presents a higher time for Load Centrality and Eccentricity.



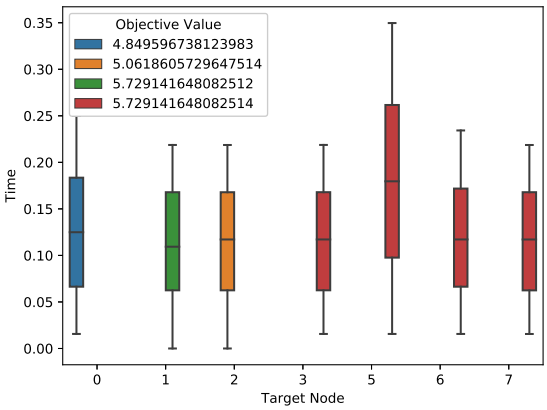
(a) Boxplot of Degree Distribution



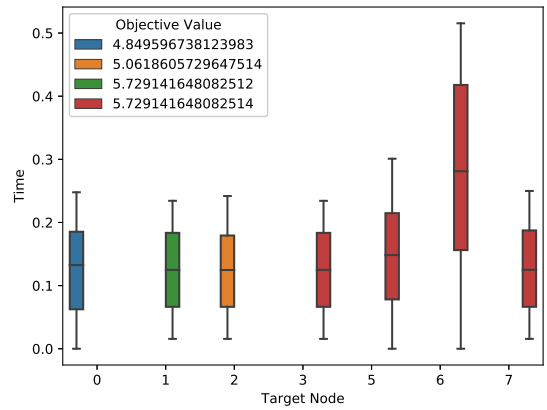
(b) Boxplot of Clustering COmponents



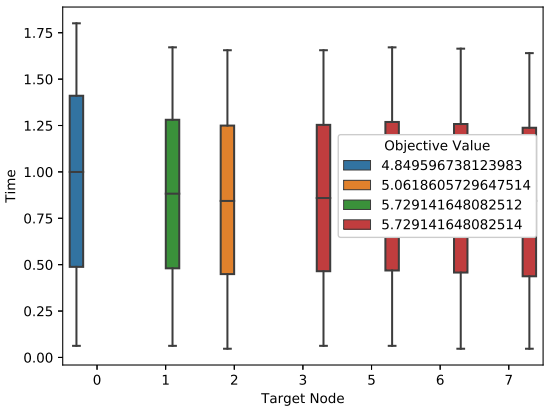
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



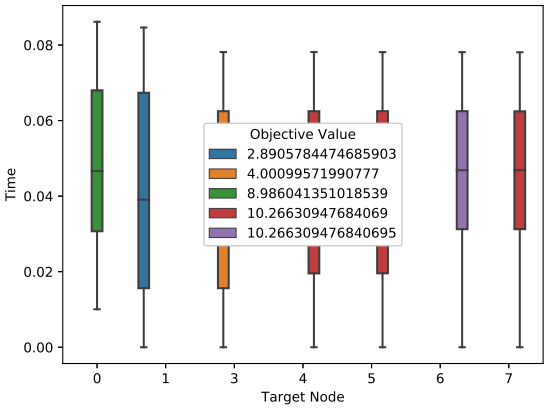
(e) Boxplot of Eccentricity



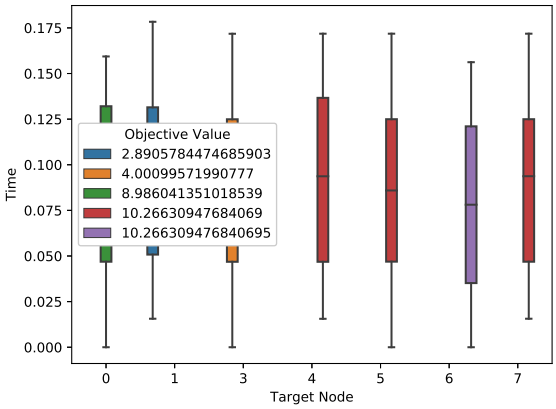
(f) Boxplot of Page Rank

Figure 5: Boxplots of Instance 04

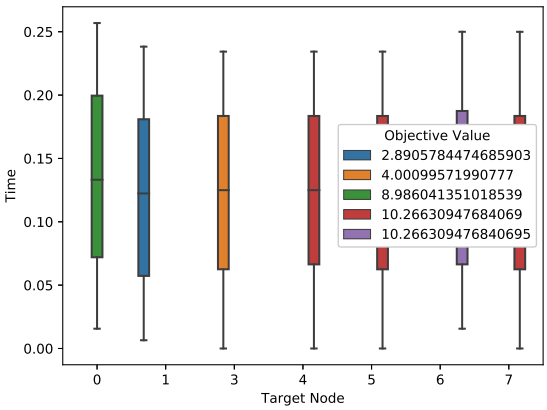
For Instance 05 all data seems to be similar without any node standing out from others as the figure shows below.



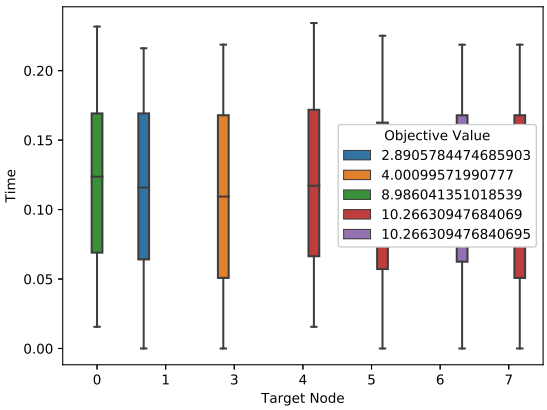
(a) Boxplot of Degree Distribution



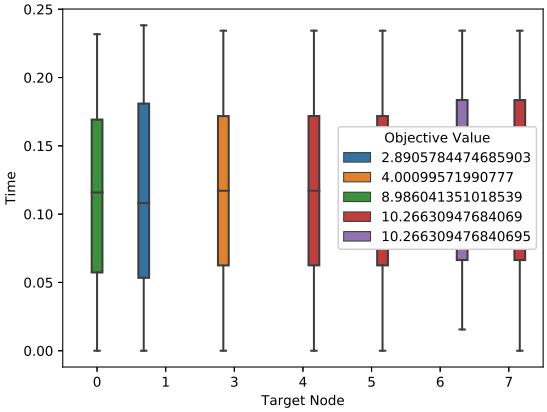
(b) Boxplot of Clustering COmponents



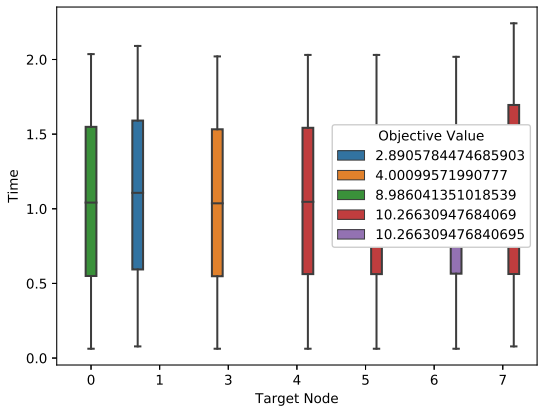
(c) Boxplot of Closeness Centrality



(d) Boxplot of Load Centrality



(e) Boxplot of Eccentricity



(f) Boxplot of Page Rank

Figure 6: Boxplots of Instance 05

As a conclusion it can be said that the maximum flow value obtained is 20.65 units with Instance 01 being node 7 the target and node 5 the source. On the other hand Instance 03 is the one with lowest objective values with nodes 2 and six Indicating they are not good target and source nodes for maximizing flow.
Page Rank and Closeness centrality are the slowest characteristics in terms of execution time, whereas the fastest characteristics are Degree distribution and Clustering Components but it cannot be say there is a correlation between the objective value and the execution time. So, aumenting the number of instances, more accurates results could be reached.

References

[1] Paul Erdös. Graph theory and probability. *Canadian Journal of Mathematics*, 11:34–38, 1959.

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

[3] Giorgio Gallo, Michael D Grigoriadis, and Robert E Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.

[4] NetworkX. Source code for networkx.drawing.layout, 2018. `https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html`, Last accessed on 2019-03-31.

[5] 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.

Homework Assignment 6: Network Flows

5273

Introduction

A water distribution system has a significant role in preserving and providing a desirable life to people, so the reliability of the supply is a key factor. Several works have defined the reliability of water distribution system [3, 4, 6] as the probability or ability of the system to meet the demands of consumers during a specified time. The demands are specified in terms of the flow rates at an adequate range of pressure.
The problem of determining connectivity of a network arises in issues of network design and reliability. In a network with random edge failures, the network could be partitioned at the minimum cuts. The minimum cut problem consists in given an undirected graph with n vertices and m edges, and it is need a partition of the vertices into two nonempty sets so as to minimize the number of edges crossing between them [5]. Su et al. [8] defines the minimum cut set as a set of system components(i.e., pipes) which, when failed, causes failure of the whole system. On the other hand, system failure will not happen if any of this set of components does not fail [2].
In this work it is studied a water distribution pipeline network throughout a connectivity analysis. It could be useful to have an idea of the reliability of the network, applying a minimum cut-set method [1], which allows the calculation of nodal pressures.
For this project the library NetworkX of Python has been used to generate an undirected powerlaw cluster graph of 121 nodes and 120 edges [7]. Edge weights have been generated randomly with non-negative values. This work it is run on an Intel Celeron CPU @ 1.10 GHz with 4 GB RAM laptop.

Case of Study: Water Distribution System

It is presented a case about a Water Distribution Supplier (WDS), represented as node 0. The system is composed of approximately 120 pipes and 121 demand nodes (corresponding to users) that are spread across an area. To analyze the interrelationship among the components, the system is first transformed into a network repesentation.

```
G=nx.balanced_tree(3,4)

n_nodes=len(G)
n_edges=nx.number_of_edges(G)

print("Number of nodes:",n_nodes)
print("Number of edges:",n_edges)

#Assign normally distributed weights to edges:
weights = np.random.normal(3, 0.5, nx.number_of_edges(G))
w = 0
for u, v, d in G.edges(data=True):
    d['weight'] = weights[w]
    w += 1

#Change node properties:
color_map = []
node_sizes = []
for node in G:
    if node == 0:
        color_map.append('skyblue')
        node_sizes.append(900)
    else:
        color_map.append('red')
        node_sizes.append(200)
```

graphs_t6.py

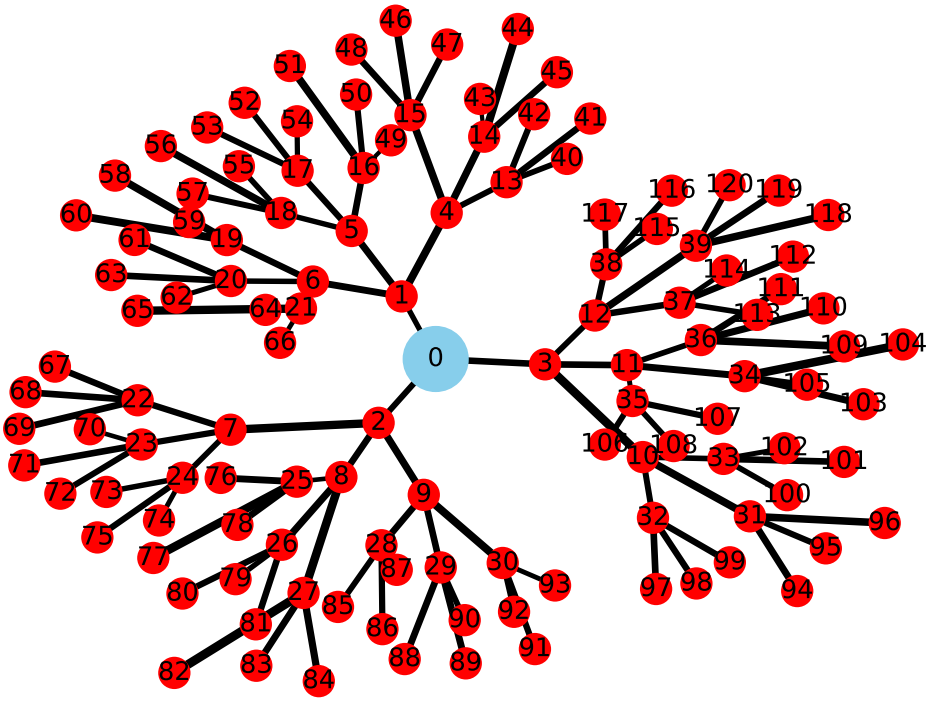


Figure 1: Water distribution network

Figure 1 shows this graph representation where can be seen 3 different clusters. This groups represent the neighbourhoods where water needs to be transported.
For Cluster 1 is run the maximum flow algorithm, collecting data of the flow values. The source node is always WDS and the target is the demanding node of that cluster. This algorithm is applied to every node of the cluster. The data has a mean of 2.61 units with a standard deviation of 0.474. With these flow values a histogram is constructed.

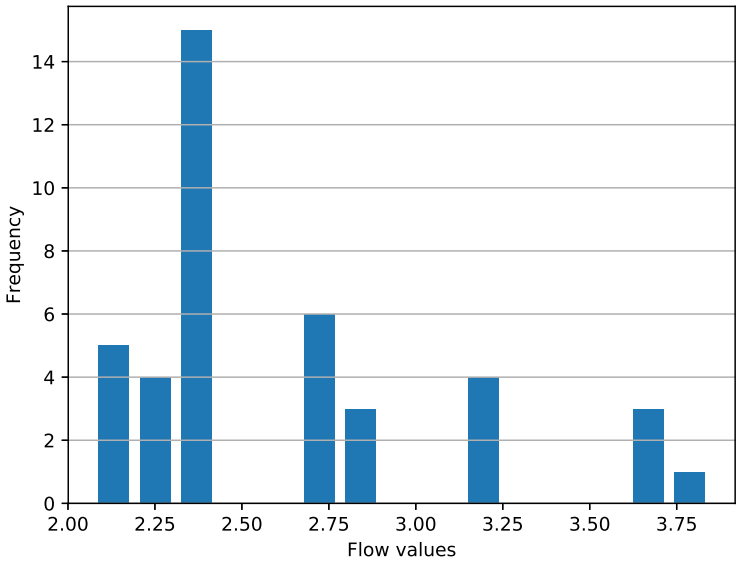


Figure 2: Histogram of water flow values for Cluster 1

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a $p - value$ of 9.42×10^{-5} so the data is not normally distributed.

```
# Analysis for Cluster 1:
for i in cluster1:
    #Maximum flow algorithm:
    flow_value,flow_dict = nx.maximum_flow(G, 0, i, capacity='weight')
    c1_values.append(flow_value)

    df=pd.DataFrame({'Cluster':[1],
                     'Flow_Value':flow_value})

    all_data=all_data.append(df)

mean=np.mean(c1_values)
std_dev=np.std(c1_values)

normality_test=stats.shapiro(c1_values)

print("Mean for Cluster 1:",mean)
print("Standard deviation for Cluster 1:",std_dev)
print("Normality test for Cluster 1:",normality_test,"\n")

#Histogram for cluster 1:
hist, bin_edges=np.histogram(c1_values,density=True)
first_edge, last_edge = np.min(c1_values),np.max(c1_values)

n_equal_bins = 15
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(c1_values,bins=bin_edges,rwidth=0.75)
plt.xlabel('Flow values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.savefig("Histogram_Cluster1.eps", format="EPS")
plt.show(1)
```

graphs_t6.py

For Cluster 2 is run the maximum flow algorithm, collecting data of the flow values. The source node is always the WDS and the target is the demanding node of that cluster. This algorithm is applied to every node of the cluster. The data has a mean of 2.54 units with a standard deviation of 0.219. With these flow values a histogram is constructed.

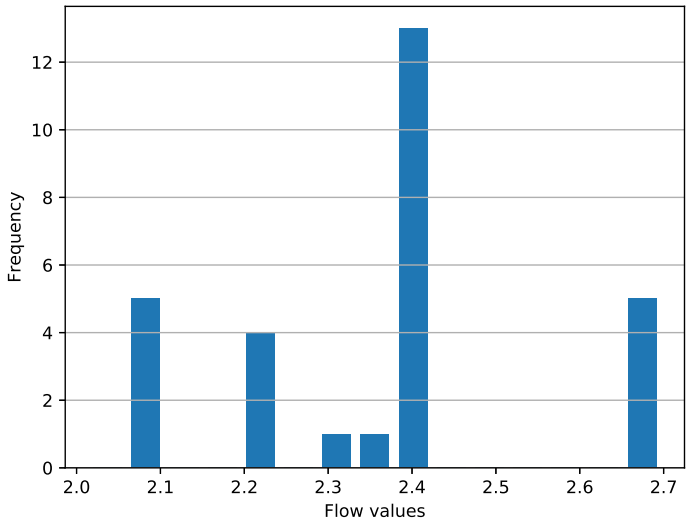


Figure 3: Histogram of water flow values for Cluster 2

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a $p - value$ of 2.90×10^{-7} so the data is not normally distributed.

```python
# Analysis for Cluster 2:
for j in cluster2:
    # Maximum flow algorithm:
    flow_value, flow_dict = nx.maximum_flow(G, 0, j, capacity='weight')
    c2_values.append(flow_value)

    df = pd.DataFrame({'Cluster': [2],
                       'Flow_Value': flow_value})

    all_data = all_data.append(df)

mean=np.mean(c2_values)
std_dev=np.std(c2_values)

normality_test=stats.shapiro(c2_values)

print("Mean for Cluster 2:",mean)
print("Standard deviation for Cluster 2:",std_dev)
print("Normality test for Cluster 2:",normality_test,"\n")

#Histogram of Cluster 2
hist, bin_edges=np.histogram(c2_values,density=True)
first_edge, last_edge = np.min(c2_values),np.max(c2_values)

n_equal_bins = 15
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(c1_values,bins=bin_edges,rwidth=0.75)
plt.xlabel('Flow values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.savefig("Histogram_Cluster2.eps", format="EPS")
plt.show(2)
```

graphs_t6.py

For Cluster 3 is run the maximum flow algorithm, collecting data of the flow values. The source node is always WDS and the target is the demanding node of that cluster. This algorithm is applied to every node of the cluster. The data has a mean of 2.79 units with a standard deviation of 0.35. With these flow values a histogram is constructed.

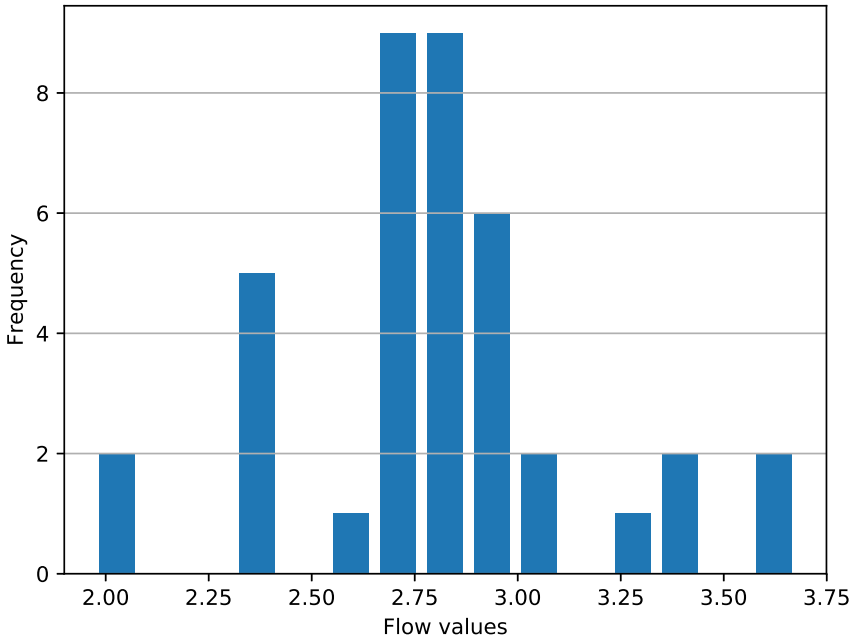


Figure 4: Histogram of water flow values for Cluster 3

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a $p - value$ of 0.044 so the data is not normally distributed.

```
hist , bin_edges=np.histogram(c3_values ,density=True)
first_edge , last_edge = np.min(c3_values),np.max(c3_values)

n_equal_bins = 15
bin_edges = np.linspace(start=first_edge , stop=last_edge ,num=n_equal_bins + 1, endpoint=True)

plt.hist(c3_values ,bins=bin_edges ,rwidth=0.75)
plt.xlabel('Flow values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.savefig("Histogram_Cluster3.eps", format="EPS")
plt.show(3)
```

graphs_t6.py

In order to compare the demanding amount of water of every cluster a boxplot is constructed. Here we can see Cluster 3 is a bit more water demanding than the others. Cluster 1 is the one with more dispersion and also have the outlier with higuest flow value of all clusters.

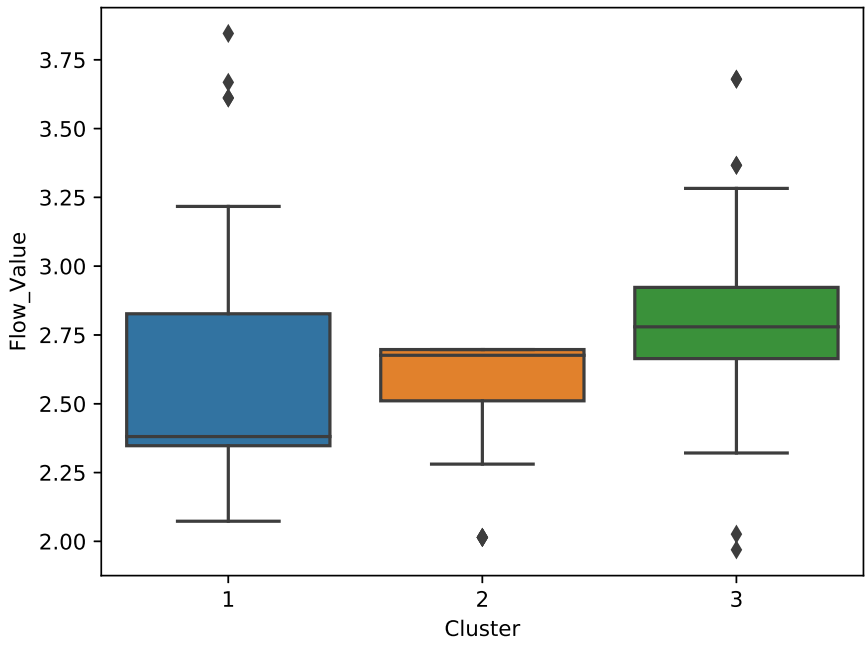


Figure 5: Boxplot of water flow values for all Clusters

The minimum cut set approach could be applied to calculate this system reliability. The impact of link failures on source-demand connectivity it could be a measure of the impact of the mechanical reliability of the water distribution network [9]. The focus is on whether a demand node can get any water from the available sources. After the cut the network is divided in to sets of nodes:

Set A	Set B
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111}	{37, 38, 39, 12, 112, 113, 114, 115, 116, 117, 118, 119, 120}

Table 1: Sets of nodes after the minimum cut.

The cut is performed at the pipes with lower demands (lower edge weights). Here the minimum cut value is the sum of the capacities of the cut edges. This value when the algorithm is executed is 2.664 units.

References

[1] Muhammad A Al-Zahrani and Juned Laiq Syed. Evaluation of municipal water distribution system reliability using minimum cut-set method. *Journal of King Saud University-Engineering Sciences*, 18(1):67–81, 2005.

[2] Roy Billinton and Ronald Norman Allan. *Reliability evaluation of engineering systems*. Springer, 1992.

[3] M John Cullinane, Kevin E Lansey, and Larry W Mays. Optimization-availability-based design of water-distribution networks. *Journal of Hydraulic Engineering*, 118(3):420–441, 1992.

[4] Ian Goulter. Analytical and simulation models for reliability analysis in water distribution systems. In *Improving efficiency and reliability in water distribution systems*, pages 235–266. Springer, 1995.

[5] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.

[6] Arnold Kaufmann, Roger Cruon, and Daniel Grouchko. *Mathematical Models for the Study of the Reliability of Systems*. Elsevier, 1977.

[7] NetworkX. Source code for networkx.drawing.layout, 2018. `https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html`, Last accessed on 2019-05-10.

[8] Yu-Chun Su, Larry W Mays, Ning Duan, and Kevin E Lansey. Reliability-based optimization model for water distribution systems. *Journal of Hydraulic Engineering*, 113(12):1539–1556, 1987.

[9] Shu-Li Yang, Nien-Sheng Hsu, Peter WF Louie, and William WG Yeh. Water distribution network reliability: Connectivity analysis. *Journal of Infrastructure Systems*, 2(2):54–64, 1996.