# ComplexNetworks - DM

October 18, 2018

## 1 Complex Networks - DM

Avel Guénin--Carlut
M2R Systèmes Complexes
18 octobre 2018

### 1.1 Setup

Run the "graph" library, corrected as instructed in the 29/09 mail, and supplemented as instructed.

Just so you know, all methods are defined in the library and loaded here. The code I show you below as defining the different methods I was instructed to code is only copied from the library so that you can judge my work from the notebook only.

I also imported reload (to be able to work on the notebook and the library parallely) ; deepcopy (to be able to work on altered versions of G without altering G itself, as it will be easier for you to know what results are to be expected from G) ; randrange (for building random graphs on which to test methods) ; loadtxt (for data importation)

```
In [1]: from importlib import reload
        from copy import deepcopy
        from random import randrange, sample
        from numpy import loadtxt

        import graph as g
        reload(g)
```

```
Out[1]: <module 'graph' from '/home/avel/Documents/Travail/2018-19 : M2R Systèmes Complexes/20
```

Define an arbitrary graph "G", which will be used to test the functions

```
In [2]:     d = {
              "a": ["c", "d", "g"],
              "b": ["c", "f"],
              "c": ["a", "b", "d", "f"],
              "d": ["a", "c", "e", "g"],
              "e": ["d"],
              "f": ["b", "c"],
              "g": ["a", "d"]
```

```
}

G = g.Graph(d)
```

## 1.2 1 - Building an elementary graph library

### 1.2.1 a) Define and test _generate_edges

A method extracting the list of a graph's edges, coded as sets of two vertices.

As the original code does not forbid to define graphs with loops or asymetrical connections, this method also checks that the graph is properly undirected and without loops. If so, it will return nothing and print an error signal instead of returning the edges set. I chose not to return the properly defined edge set, as an unproperly defined graph will mean we have to revise the code anyway.

I'd prefer to code this in the graph initialisation - but I wanted not to alter the original code.

In [3]:
```
def __generate_edges(self):
    edges = set() # I use a set structure for easy removing of doublons.

    for v1 in self.__graph_dict :
        for v2 in self.__graph_dict[v1] :

            ## I check for errors in graph definition
            if v1 == v2: # First error case : loop
                print("error : a loop is present in the graph")
                return()
            elif not (v1 in self.__graph_dict[v2]) : # Second error case : asymetr
                print("error : an edge is asymetrical")
                return()

            ## if no error exist I add the edge to the output
            else:
                edges.add(frozenset([v1,v2])) # edges are converted to frozenset b

    ### Then I convert output to a list(set) for readability
    edges_list = []
    for e in edges:
        edges_list.append(set(e))
    return edges_list
```

The function is tested by calling it on g, and on a graph containing a loop. It is called indirectly through the method "edges".

In [4]:
```
print("Edges of g")
print(G.edges())

d = {"a" : ["a"]}
loop = g.Graph(d)
print("\nEdges of a loop")
```

```
        loop.edges()

        d = {
            "a" : ["b"],
            "b" : []
        }
        directed = g.Graph(d)
        print("\nEdges of an undirected graph")
        directed.edges()
```

```
Edges of g
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'
```

```
Edges of a loop
error : a loop is present in the graph

Edges of an undirected graph
error : an edge is asymetrical
```

```
Out[4]: ()
```

As we see, the correct list of edges is displayed, and an error is correctly signaled for both problematic cases

### 1.2.2 b) Define and test add_vertex

A method adding an isolated vertex to a graph. An warning message is displayed if the vertex is already present.

```
In [5]:     def add_vertex(self, vertex):
                if vertex in self.__graph_dict:
                    print("The vertex " + vertex + " already was in the graph !")
                else :
                    self.__graph_dict[vertex]=[]
```

The function is tested by trying to add two new vertex to a new graph G1 equal to G, in both case where the vertex is originally absent or present from the graph.

```
In [6]: G1 = deepcopy(G)

        print("Original vertices and edges in the graph")
        print(G1.vertices())
        print(G1.edges())

        print("\nLet's add a new vertex")
        G1.add_vertex("h")
        print(G1.vertices())
        print(G1.edges())
```

```
        print("\nLet's add a preexisting vertex")
        G1.add_vertex("h")

        print(G1.vertices())
        print(G1.edges())
```

```
Original vertices and edges in the graph
['a', 'b', 'c', 'd', 'e', 'f', 'g']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'

Let's add a new vertex
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'

Let's add a preexisting vertex
The vertex h already was in the graph !
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'
```

As we can see, a new vertex can be added without affecting the edges, and if the vertex already was present an message is displayed and nothing else happens.

### 1.2.3 c) Define and test add_edges

A method adding an edge to the graph. Without specific instructions, I assume that the graph structure do not allow for two vertices to be linked by several edges as this is the case we studied in our classes.

A warning message is displayed if the edge links more or less than 2 elements, if it already existed, if it is a loop, or if it implies an non-existing vertex, and inconsistant instruction are not taken into account.

```
In [7]:     def add_edge(self, edge):
                """ assumes that edge is of type set, tuple or list. No loops or
                multiple edges. To complete."""

                ## Early conversion to a set assures that the function works the same on diffe
                edge = set(edge)

                ## first I check edge size
                if (len(edge) > 2) :
                    print("Error : edge has more than two elements !")
                    return()

                elif (len(edge) == 1):
                    print("Error : edge is a loop !")
                    return()
```

```
        elif (len(edge) < 1):
            print("Error : edge is empty !")
            return()

        ## Then I extract vertices
        v1 = edge.pop()
        v2 = edge.pop()


        ## Then I check that the edge is properly defined
        if set([v1,v2]) in self.edges():
            print("The edge " + str([v1,v2]) + " already existed")
            return()

        elif not ((v1 in self.vertices()) and (v2 in self.vertices())):
            print("error : undefined vertex !")
            return()

        ## if no error can be detected, I add the edge as two symmetrical entries in t
        self.__graph_dict[v1].append(v2)
        self.__graph_dict[v2].append(v1)

        return()
```

Again, we test the method in different cases. Different data type are not tested here for compacity, but early conversion to a set insure that the method behaviour is homogenous.

```
In [8]: G1 = deepcopy(G)

        print("Original vertices and edges in the graph")
        print(G1.vertices())
        print(G1.edges())

        print("\nLet's add {} (empty edge)")
        G1.add_edge({})
        print(G1.vertices())
        print(G1.edges())

        print("\nLet's add {'a','a'} (a loop)")
        G1.add_edge({"a","a"})
        print(G1.vertices())
        print(G1.edges())

        print("\nLet's add {'a','b','c'} (a multiple vertices edge)")
        G1.add_edge({"a","b","c"})
        print(G1.vertices())
        print(G1.edges())
```

```
        print("\nLet's add a proper new edge {'a','b'}")
        G1.add_edge({"a","b"})
        print(G1.vertices())
        print(G1.edges())

        print("\nLet's add {'a','b'} AGAIN (preexisting edge)")
        G1.add_edge({"a","b"})
        print(G1.vertices())
        print(G1.edges())

        print("\nLet's add {'a','toto'} (undefined vertices)")
        G1.add_edge({"a","toto"})
        print(G1.vertices())
        print(G1.edges())
```

```
Original vertices and edges in the graph
['a', 'b', 'c', 'd', 'e', 'f', 'g']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'

Let's add {} (empty edge)
Error : edge is empty !
['a', 'b', 'c', 'd', 'e', 'f', 'g']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'

Let's add {'a','a'} (a loop)
Error : edge is a loop !
['a', 'b', 'c', 'd', 'e', 'f', 'g']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'

Let's add {'a','b','c'} (a multiple vertices edge)
Error : edge has more than two elements !
['a', 'b', 'c', 'd', 'e', 'f', 'g']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'

Let's add a proper new edge {'a','b'}
['a', 'b', 'c', 'd', 'e', 'f', 'g']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'

Let's add {'a','b'} AGAIN (preexisting edge)
The edge ['a', 'b'] already existed
['a', 'b', 'c', 'd', 'e', 'f', 'g']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'

Let's add {'a','toto'} (undefined vertices)
error : undefined vertex !
['a', 'b', 'c', 'd', 'e', 'f', 'g']
[{'a', 'g'}, {'b', 'c'}, {'b', 'f'}, {'a', 'd'}, {'f', 'c'}, {'g', 'd'}, {'e', 'd'}, {'a', 'c'
```

We can see that the method works properly, and returns correct error signals.

# 2 2 - Methods related to degree

## 2.1 2.1 - Degree and isolated vertices

### 2.1.1 a) Define and test vertex_degree

A method returning the degree of all vertices in a dictionary

```
In [9]:     def vertex_degree(self):
                degree = dict()
                for v in self.__graph_dict.keys():
                    degree[v] = len(self.__graph_dict[v])

                return degree
```

Let's try it on G, and also try it with an additional isolated vertex :

```
In [10]: print(G.vertex_degree())

         G1 = deepcopy(G)
         G1.add_vertex("h")
         print(G1.vertex_degree())
```

```
{'a': 3, 'b': 2, 'c': 4, 'd': 4, 'e': 1, 'f': 2, 'g': 2}
{'a': 3, 'b': 2, 'c': 4, 'd': 4, 'e': 1, 'f': 2, 'g': 2, 'h': 0}
```

We can see the method work, including in the limit case of isolated vertices

### 2.1.2 b) find_isolated_vertices

Method returning all isolated vertices in a set. As we have the vertex_degree method, we just have to find the keys returning 0 as a value.

```
In [11]:     def find_isolated_vertices(self):
                degree = self.vertex_degree()
                isolated_vertices=set([v for v in degree if degree[v] == 0])

                return isolated_vertices
```

```
In [12]: print(G.find_isolated_vertices())

         G1 = deepcopy(G)
         G1.add_vertex("h")
         print(G1.find_isolated_vertices())
         G1.add_vertex("i")
         print(G1.find_isolated_vertices())
```

```
set()
{'h'}
{'h', 'i'}
```

The method works for 0, 1, and 2 isolated vertices in the graph we tested.

### 2.2 2.2 - Density calculation

A method returning the density of the graph, defined as the ratio of edges number over the maximum of possible number of edges knowing the number of vertices.

As loops are not allowed, we expect the method to return m/(n*(n-1)/2) where m is the number of edges and n the number of vertices in the graph. I do not allow for density calculation for graph whose size is inferior to 2, since the maximum number of edges is then zero.

```
In [13]:     def density(self):
                 n = len(self.vertices())
                 m = len(self.edges())

                 # to avoid limit problem, I do not allow density calculation when the number
                 if n < 2 :
                     print("No density can be computed for graph with n<2")
                     return()

                 return 2*m/(n*(n-1))
```

We'll first test the limit cas of n = 0,1,2. The code is expected to display an error message and return nothing in the first two cases, but to return a (correct) value in the last two ones.

```
In [14]: print("Density of a void graph")
         G1 = g.Graph(dict())
         print(G1.density())

         print("\n")
         print("Density of an univertex graph")
         G1.add_vertex("a")
         print(G1.density())

         print("\n")
         print("Density of a bivertex graph (empty)")
         G1.add_vertex("b")
         print(G1.density())

         print("\n")
         print("Density of a bivertex graph (complete)")
         G1.add_edge(["a","b"])

         print(G1.density())
```

```
Density of a void graph
No density can be computed for graph with n<2
()


Density of an univertex graph
No density can be computed for graph with n<2
()


Density of a bivertex graph (empty)
0.0


Density of a bivertex graph (complete)
1.0
```

Then we'll test a random graph build with known density (uniform over the range of possible densities)

```
In [15]: def randgraph(n,m,label="n"):
             G1 = g.Graph(dict())

             ## I add a random, known number of vertices
             for j in range(n):
                 G1.add_vertex(label + str(j))

             ## I add a random, known, number of edges selected over all potential edges
             potential_edges=set()
             for v1 in G1.vertices():
                 for v2 in G1.vertices():
                     if not (v1 == v2):
                         potential_edges.add(frozenset([v1,v2]))

             for l in range(m):
                 e = sample(potential_edges,1)[0] # sample is used to have true-ish randomness
                 potential_edges.remove(e)
                 G1.add_edge(set(e))

             return G1



         n = randrange(10,20)
         m = randrange(n*(n-1)/2)
```

```
G1 = randgraph(n,m)
n = len(G1.vertices())
m = len(G1.edges())

d_def = 2*m/(n*(n-1))
d_method = G1.density()
print("Graph's density at definition : " + str(d_def))
print("Graph's computed density : " + str(d_method))
print("Is the computed density correct ? " + str(d_def == d_method))
```

```
Graph's density at definition : 0.7435897435897436
Graph's computed density : 0.7435897435897436
Is the computed density correct ? True
```

### 2.3 2.2 - Degree sequence

This method returns all of the vertices' degree as an ordered tuple. From the method vertex_degree, the implementation is pretty straightforward.

```
In [16]:     def degree_sequence(self):
                 degree = self.vertex_degree()
                 deg_seq = list(degree.values())
                 deg_seq.sort(reverse=True)
                 deg_seq = tuple(deg_seq)
```

Let's test it on G, and also on an empty graph, and also on a graph with only one (isolated) vertex :

```
In [17]: print(G.degree_sequence())
         G1 = g.Graph(dict())
         print(G1.degree_sequence())
         G1.add_vertex("a")
         print(G1.degree_sequence())
```

```
(4, 4, 3, 2, 2, 2, 1)
()
(0,)
```

Well, it all seem to work.

### 2.4 2.4 - Erdös-Gallai theorem

This method returns from a tuple a Boolean value indicating whether there exist a graph with this sequence as a degree sequence, using the Erdös-Gallai theorem.

As instructed, it is coded as a method inside the "Graph" class and not as a standalone.

```
In [18]:     def erdos_gallai(self,seq):
                 n = len(seq)

                 ## check non-increase condition
                 for i in range(1,n):
                     if seq[i]>seq[i-1]:
                         print("Sequance isn't non increasing")
                         return(False)

                 ## check sum parity
                 if sum(seq)%2 == 1:
                     print("Sequence has odd sum")
                     return(False)

                 ## check inequality for all k
                 for k in range(1,n):
                     l_term = sum(seq[0:k])

                     r_term = k*(k-1)
                     r_term += sum(min(list(seq[k+1:]),[k for i in range(k+1,n)]))

                     if l_term > r_term:
                         print("Inequality invalid for k = " + str(k))
                         return(False)

                 ## and if no condition is invalid
                 return(True)
```

Let's test it on actual degree sequence from G and a random graph, and on sequence we build to falsify each condition independantly. Let's also check it does not crash on the limit case of a null sequence.

```
In [19]: print("Erdös-Gallai method for G's degree sequence ? ")
         seq = G.degree_sequence()
         print(G.erdos_gallai(seq))

         n = randrange(10,20)
         m = randrange(n*(n-1)/2)
         G1 = randgraph(n,m)

         print("Erdös-Gallai method for a random graph's degree sequence ? ")
         seq = G1.degree_sequence()
         print(G1.erdos_gallai(seq))
         print("\n")

         print("Erdös-Gallai method for a odd sum sequence ? ")
         seq = seq.__add__((1,))
         print(G1.erdos_gallai(seq))
```

```
        print("\n")

        print("Erdös-Gallai method for a locally increasing sequence ? ")
        seq = seq.__add__((3,))
        print(G1.erdos_gallai(seq))
        print("\n")

        print("Erdös-Gallai method for a sequence falsifying the inequality ? ")
        seq = (8,3,1)
        print(G1.erdos_gallai(seq))
        print("\n")

        print("Erdös-Gallai method for a null sequence ? ")
        seq = ()
        print(G1.erdos_gallai(seq))
```

```
Erdös-Gallai method for G's degree sequence ?
True
Erdös-Gallai method for a random graph's degree sequence ?
True


Erdös-Gallai method for a odd sum sequence ?
Sequence has odd sum
False


Erdös-Gallai method for a locally increasing sequence ?
Sequance isn't non increasing
False


Erdös-Gallai method for a sequence falsifying the inequality ?
Inequality invalid for k = 1
False


Erdös-Gallai method for a null sequence ?
True
```

### 2.5 2.5 - Global clustering coefficient

A method returning the global clustering coefficient of a graph.

```
In [20]:     def global_clustering_coefficient(self):
                 ## Let's first build the set of all triplets in the graph
                 global_triplets = set()
```

```
                for v in self.vertices():
                    local_triplets = set()
                    for i in range(len(self.__graph_dict[v])):
                        for j in range(i+1,len(self.__graph_dict[v])):
                            s = frozenset([v,self.__graph_dict[v][i],self.__graph_dict[v][j]])
                            local_triplets.add(s)

                    for t in local_triplets :
                            global_triplets.add(t)

                ## Then let's count the number of triplets that are triangles
                t_counter = 0
                for s in global_triplets:
                    [v1,v2,v3] = list(s)
                    if (v1 in self.__graph_dict[v2]) and (v2 in self.__graph_dict[v3]) and (v3
                        t_counter+=1

                ## ...and return the result. By convention, we will take 0 for all graph where
                if len(global_triplets) > 0 :
                    return(t_counter/len(global_triplets))
                else:
                    return(0)
```

We check below on G, a complete graph, an edgeless graph, and a void graph (in the latter case we expect the function to return 0)

```
In [21]: print("density of G : " + str(G.global_clustering_coefficient()))

         n = randrange(10,20)
         m = int(n*(n-1)/2)
         G1 = randgraph(n,m)
         print("density of a complete graph : " + str(G1.global_clustering_coefficient()))


         n = randrange(10,20)
         m = 0
         G1 = randgraph(n,m)
         print("density of an edgeless graph : " + str(G1.global_clustering_coefficient()))

         G1 = g.Graph(dict())
         print("density of an empty graph : " + str(G1.global_clustering_coefficient()))
```

```
density of G : 0.25
density of a complete graph : 1.0
density of an edgeless graph : 0
density of an empty graph : 0
```

Everything seems to be working.

# 3 3 - Methods related to graph traversal

## 3.1 3.1 - Connected components

A method returning all connected components as a list of set. For the sake of generality, it does not return directly number and size of all connected components, but these quantities can be straightforwadly computed from the method's output as shown in the tests below.

In [22]:
```
    def connected_component(self):

        ## Initialisation
        visited = dict() # this store both the set of visited vertices and the order
        count = 0 # this is the variable counting visit order
        connected_components_list = []

        for v in self.vertices():
            visited[v] = count # 0 is understood as "not visited"


        ## Calculation
        while (0 in visited.values()):

            ## Initialisation of a single connected component's exploration

            adj_vertices = set() # contains the list of vertices that are adjacent to
            connected_component = set() # contains the set of elements of the connect

            v = [key  for (key, value) in visited.items() if value == 0][0] # pick an
            adj_vertices.add(v)

            ## explore until end of connected component
            while len(adj_vertices) > 0:
                v = adj_vertices.pop()
                connected_component.add(v)
                l = [v_adj for v_adj in self.__graph_dict[v] if visited[v_adj] == 0]
                adj_vertices |= set(l)

                count += 1
                visited[v] = count

            ## add connected component to connected component list
            connected_components_list.append(frozenset(connected_component)) # frozen

        return(connected_components_list)
```

Let's test it in a number of simple cases where we know the expected result and can quickly check consistency, and also in the case of a random graph.

In [23]:
```
print("Connex components of G")
C = G.connected_component()
```

```
print(C)
print("Number of connected components : " + str(len(C)))
print("Size of connected components : " + str([len(s) for s in C]))
print("\n")


print("Connex components of a complete graph")
n = randrange(10,20)
m = int(n*(n-1)/2)
G1 = randgraph(n,m)
C = G1.connected_component()
print(C)
print("Number of connected components : " + str(len(C)))
print("Size of connected components : " + str([len(s) for s in C]))
print('\n')


print("Connex components of a edgeless graph")
n = randrange(10,20)
m = 0
G1 = randgraph(n,m)
C = G1.connected_component()
print(C)
print("Number of connected components : " + str(len(C)))
print("Size of connected components : " + str([len(s) for s in C]))
print("\n")


print("Connex components of an empty graph")
G1 = g.Graph(dict())
C = G1.connected_component()
print(C)
print("Number of connected components : " + str(len(C)))
print("Size of connected components : " + str([len(s) for s in C]))
print("\n")


print("Connex components of a random graph (density uniform over all possible densitie
n = randrange(10,20)
m = randrange(n*(n-1)/2)
G1 = randgraph(n,m)
C = G1.connected_component()
print(C)
print("Number of connected components : " + str(len(C)))
print("Size of connected components : " + str([len(s) for s in C]))
print("\n")
```

```
Connex components of G
```

```
[frozenset({'e', 'c', 'a', 'f', 'b', 'g', 'd'})]
Number of connected components : 1
Size of connected components : [7]


Connex components of a complete graph
[frozenset({'n7', 'n8', 'n9', 'n3', 'n6', 'n0', 'n13', 'n2', 'n10', 'n11', 'n1', 'n4', 'n12',
Number of connected components : 1
Size of connected components : [14]


Connex components of a edgeless graph
[frozenset({'n0'}), frozenset({'n1'}), frozenset({'n2'}), frozenset({'n3'}), frozenset({'n4'})
Number of connected components : 16
Size of connected components : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]


Connex components of an empty graph
[]
Number of connected components : 0
Size of connected components : []


Connex components of a random graph (density uniform over all possible densities)
[frozenset({'n18', 'n7', 'n14', 'n9', 'n3', 'n13', 'n2', 'n10', 'n15', 'n1', 'n4', 'n17', 'n8'
Number of connected components : 1
Size of connected components : [19]
```

Everything is OK.

### 3.2 3.2 - Shortest path

A method returning as a dict of dicts the distance between all pairs of vertices. We use for this purpose a BFS algorithm. As we are not instructed to compute shortest pathes but only distance, I only keep track of the distance from the active origin node and not of each node's parent.

```
In [24]:     def shortest_path(self):
                 distances = dict()

                 ## Distances initialisation
                 for v1 in self.vertices():
                     distances[v1] = dict()
                     for v2 in self.vertices():
                         distances[v1][v2] = float("inf")
```

```
            ## definition of origin node for the BFS
            for v_ini in self.vertices():
                d = 0
                adj_vertices = set() ## vertices that are adjacent to the currently explo
                adj_vertices.add(v_ini)

                ## explore with BFS
                while len(adj_vertices) > 0:
                    d_class = [v for v in adj_vertices] ## distance class

                    ## vertices closest to origine are explored first
                    for v in d_class :
                        distances[v_ini][v] = d
                        adj_vertices.remove(v)

                    ## then we add their child to the adjacent set
                    for v1 in d_class :
                        adj_vertices |= set([v2 for v2 in self.__graph_dict[v1] if distan
                    d+=1

            return(distances)
```

We'll check that proper results are returned on G, and also that the output respects null distance to self, symmetry, and triangular inequality on a random graph.

```
In [25]: print("Distances in G")
         print(G.shortest_path())
         print("\n_____________\n")


         n = randrange(10,20)
         m = randrange(n*(n-1)/2)
         G1 = randgraph(n,m)
         dist = G1.shortest_path()
         print("Distances computed in a random graph")
         print("\n")

         print("Is null distance to self respected ?")
         result = "Yes it is"
         for v in G1.vertices():
             if not dist[v][v] == 0:
                 result = "No it is not"
                 break
         print(result)
         print("\n")

         print("Is symmetry respected ?")
         result = "Yes it is"
```

```
        for v1 in G1.vertices():
            for v2 in G1.vertices():
                if not (dist[v1][v2] == dist[v2][v1]):
                    result = "No it is not"
                    break
        print(result)
        print("\n")


        print("Is triangular inequality respected ?")
        result = "Yes it is"
        for v1 in G1.vertices():
            for v2 in G1.vertices():
                for v3 in G1.vertices():
                    if (dist[v1][v2] > dist[v2][v3] + dist[v1][v3]):
                        result = "No it is not"
                        break
        print(result)
        print("\n")
```

```
Distances in G
{'a': {'a': 0, 'b': 2, 'c': 1, 'd': 1, 'e': 2, 'f': 2, 'g': 1}, 'b': {'a': 2, 'b': 0, 'c': 1,

--------------

Distances computed in a random graph


Is null distance to self respected ?
Yes it is


Is symmetry respected ?
Yes it is


Is triangular inequality respected ?
Yes it is
```

I also checked consistency in the simple limit cases of empty, edgeless, and complete graphs, but for compacity's sake I'll ask you to trust me on this issue.

### 3.3 3.3 - Diameter

Two methods respectively computing the diameter and biggest component diameter of the graph. Knowing the preceding method, implementation really is straightforward.

```
In [26]:     def diameter(self):
                 d_max = 0
                 distances = self.shortest_path()
                 for v1 in self.vertices():
                     for v2 in self.vertices():
                         d_max = max(distances[v1][v2],d_max)
                 return(d_max)

             def biggest_component_diameter(self):
                 d_max = 0
                 distances = self.shortest_path()
                 for v1 in self.vertices():
                     for v2 in self.vertices():
                         if float("inf") > distances[v1][v2]:
                             d_max = max(distances[v1][v2],d_max)
                 return(d_max)
```

We'll test them both on G and on a custom disjoint graph

```
In [27]: print("Diameter of G : " + str(G.diameter()) + "\n")
         print("Biggest connex component diameter of G : " + str(G.biggest_component_diameter()
         print("\n_________________\n")


         print("Let's define a custom non connex graph G1 from our randgraph method\n")
         n1 = randrange(10,20)
         m1 = randrange(n1*(n1-1)/2)
         convex_component_1 = randgraph(n1,m1)

         n2 = randrange(10,20)
         m2 = randrange(n2*(n2-1)/2)
         convex_component_2 = randgraph(n2,m2,"m")

         G1 = g.Graph(dict())
         for v in (convex_component_1.vertices()+convex_component_2.vertices()):
             G1.add_vertex(v)
         for e in (convex_component_1.edges()+convex_component_2.edges()):
             G1.add_edge(e)

         print("Diameter of G1 : " + str(G1.diameter()) + "\n")
         print("Biggest connex component diameter of G1 : " + str(G1.biggest_component_diameter
```

```
Diameter of G : 3

Biggest connex component diameter of G : 3

----------------
```

```
Let's define a custom non connex graph G1 from our randgraph method

Diameter of G1 : inf

Biggest connex component diameter of G1 : 5
```

We got the expected results, the methods seem to be working.

### 3.4 3.4 - Spanning tree

I implement the slightly more general method of spanning forests, which is equivalent in the case of a connected graph. Unlike the case of distances computation, we need to keep track of the parent of each node while we search it.

The method is designed to return a new graph rather than modify the input.

```
In [28]:     def spanning_forest(self):
                 ## Initialisation of the spanning forest as a new graph
                 spanning_forest = Graph(dict())


                 while [v for v in self.vertices() if v not in spanning_forest.vertices()]:

                     ## Initialisation of a new connected component
                     v_ini = [v for v in self.vertices() if v not in spanning_forest.vertices()
                     spanning_forest.add_vertex(v_ini)
                     v_frontier = [v_ini] # list of current spanning trees elements whose neigh

                     ## Each frontier node is used to explore its neighbours. Then it is remov
                     while v_frontier:
                         v1 = v_frontier[0]
                         v1_childs = [v for v in self.__graph_dict[v1] if v not in spanning_for
                         v_frontier.remove(v1)

                         ## Each explored node is added to the spanning tree, as well as its p
                         for v2 in v1_childs :
                             spanning_forest.add_vertex(v2)
                             spanning_forest.add_edge([v1,v2])
                             v_frontier.append(v2)

                 return(spanning_forest)
```

Let's test it on a random graph. We expect a spanning forest to have the same connected components as its origin graph, and as many edge as vertices number minus connected components number. Because of trees duality properties, it is enough to test the validity of a spanning forest.

```
In [29]: print("We build a number of random graph and its random forest\n\n")

         for i in range(10):
```

```
            n = randrange(10,20)
            m = randrange(n*(n-1)/2)
            G1 = randgraph(n,m)
            T1 = G1.spanning_forest()


            print("\nNew graph built !")
            print("Is the random graph connected ? " + str(len(G1.connected_component())==1))
            print("Are connected components identical ? " + str(set(G1.connected_component())
            print("Is there the correct number of edges in the forest ? " + str(len(T1.edges()
```

We build a number of random graph and its random forest

```
New graph built !
Is the random graph connected ? True
Are connected components identical ? True
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? True
Are connected components identical ? True
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? True
Are connected components identical ? True
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? True
Are connected components identical ? True
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? False
Are connected components identical ? True
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? False
Are connected components identical ? True
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? True
Are connected components identical ? True
```

```
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? True
Are connected components identical ? True
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? True
Are connected components identical ? True
Is there the correct number of edges in the forest ? True

New graph built !
Is the random graph connected ? True
Are connected components identical ? True
Is there the correct number of edges in the forest ? True
```

We can see that the algorithm works in both connected and disconnected cases.

# 4 4 - Testing on real datasets

## 4.1 4.1 - Importing real data

A function loading a graph from a filename. Implementation is straightforward from the nunpy.loadtxt function. The file must be in the same directory as the notebook (or a filepath must be specified)

```
In [30]: def graph_load(filename):
                 G1 = g.Graph(dict())
                 data = loadtxt(filename)

                 for e in data :
                     if str(e[0]) not in G1.vertices():
                         G1.add_vertex(str(e[0]))
                     if str(e[1]) not in G1.vertices():
                         G1.add_vertex(str(e[1]))
                     G1.add_edge([str(e[0]),str(e[1])])
                 return(G1)

In [31]: print("Loading Zachary...")
         zachary = graph_load("zachary_connected.txt")

         print("\nLoading random, n = 100...")
         random100 = graph_load("graph_100n_1000m.txt")

         print("\nLoading random, n = 1000...")
         random1000 = graph_load("graph_1000n_4000m.txt")
```

```
Loading Zachary...

Loading random, n = 100...
The edge ['5.0', '14.0'] already existed
The edge ['22.0', '13.0'] already existed
The edge ['17.0', '23.0'] already existed
The edge ['30.0', '3.0'] already existed
The edge ['34.0', '25.0'] already existed
The edge ['31.0', '35.0'] already existed
The edge ['16.0', '37.0'] already existed
The edge ['44.0', '38.0'] already existed
The edge ['15.0', '47.0'] already existed
The edge ['48.0', '20.0'] already existed
The edge ['34.0', '49.0'] already existed
The edge ['50.0', '27.0'] already existed
The edge ['1.0', '56.0'] already existed
The edge ['58.0', '33.0'] already existed
The edge ['56.0', '59.0'] already existed
The edge ['45.0', '61.0'] already existed
The edge ['61.0', '52.0'] already existed
The edge ['45.0', '62.0'] already existed
The edge ['34.0', '67.0'] already existed
The edge ['39.0', '67.0'] already existed
The edge ['69.0', '25.0'] already existed
The edge ['69.0', '47.0'] already existed
The edge ['28.0', '71.0'] already existed
The edge ['72.0', '60.0'] already existed
The edge ['73.0', '62.0'] already existed
The edge ['34.0', '74.0'] already existed
The edge ['8.0', '77.0'] already existed
The edge ['79.0', '77.0'] already existed
The edge ['81.0', '10.0'] already existed
The edge ['19.0', '86.0'] already existed
The edge ['88.0', '23.0'] already existed
The edge ['89.0', '65.0'] already existed
The edge ['33.0', '91.0'] already existed
The edge ['15.0', '92.0'] already existed
The edge ['93.0', '64.0'] already existed
The edge ['12.0', '95.0'] already existed
The edge ['97.0', '20.0'] already existed
The edge ['97.0', '77.0'] already existed
The edge ['98.0', '51.0'] already existed
The edge ['99.0', '78.0'] already existed

Loading random, n = 1000...
The edge ['441.0', '127.0'] already existed
The edge ['503.0', '421.0'] already existed
The edge ['425.0', '669.0'] already existed
```

```
The edge ['675.0', '24.0'] already existed
The edge ['648.0', '704.0'] already existed
The edge ['593.0', '706.0'] already existed
The edge ['707.0', '414.0'] already existed
The edge ['723.0', '617.0'] already existed
The edge ['900.0', '217.0'] already existed
The edge ['340.0', '904.0'] already existed
The edge ['937.0', '9.0'] already existed
```

As of now, doublons in the data cause warning messages to be displayed. To avoid this would take either a cleanup of the data (unjustified as this is, in our case, a minor inconveniance) or a check during the adding of an edge (which would cause the code to be m^2 complex, slowing it down significantly).

Note that I coded a check during the adding of a vertex, as there are significantly more doublons in vertices than in edges, and as n^2 << m^2.

Let's check now that our import was correct in the Zachary case.

```
In [32]: print(zachary.vertices())
         print(zachary.edges())
```

```
['2.0', '0.0', '3.0', '1.0', '4.0', '5.0', '6.0', '7.0', '8.0', '9.0', '10.0', '11.0', '12.0',
[{'31.0', '33.0'}, {'7.0', '4.0'}, {'1.0', '22.0'}, {'34.0', '26.0'}, {'18.0', '1.0'}, {'0.0',
```

A visual check allow us to conclude that the import was correct.

### 4.2 4.2 - Properties of supplied graphs

As I couldn't find a ready-to-use table library for Python, we'll do with a slightly more primitive data display.

```
In [33]: print("dataset  |  vertices nb  | edges nb  |  density  |  diameter  |  clustering coe

         print("Zachary" + "  |  " + str(len(zachary.vertices())) + "  |  " + str(len(zachary.e
         print("random100" + "  |  " + str(len(random100.vertices())) + "  |  " + str(len(rando
         print("random1000" + "  |  " + str(len(random1000.vertices())) + "  |  " + str(len(ran
```

```
dataset  |  vertices nb  | edges nb  |  density  |  diameter  |  clustering coefficient

Zachary  |  33  |  78  |  0.14772727272727273  |  5  |  0.0574468085106383
random100  |  100  |  960  |  0.19393939393939394  |  3  |  0.07372662595227601
random1000  |  1000  |  3989  |  0.007985985985985987  |  6  |  0.002343552064859387
```

A discrepancy can be found in the number of edges for both random graphs, but it is due to the doublons in the data.