FKIRI KHALED

2018

TP ALGORTHMES DE FOURMI

Ce sujet possède une sensibilité Opel Data. On va travailler sur les vraies données fournies par le gouvernement britanniques.
https://www.kaggle.com/getthedata/open-pubs
Le but est de montrer une tournée à vol d'oiseau pour faire la distance minimum entre les pubs.

I. Contexte

Le but de ce document et de vous expliquer en détails l'algorithme utilisé pour calculer la distance les pubs.

Pour commencer il nous faut :

- La version Python 3.3.6
- Installer le module **csv** implémente des classes pour lire et écrire des données tabulaires au format CSV.
- Installer le module **Pants** qui nous permettent de déterminer rapidement comment visiter une collection des nœuds (latitude, longitude) interconnectés de sorte que le travail effectué soit minimisé. Pants c'est l'outil que j'ai utilisé pour calculer la distance minimum des pubs.
- Installer le module **geopy** nous permet de calculer la distance 3D sur terre.
- Installer le module **time** nous permet de calculer le temps d'exécution.
- Installer le module **warnings** pour filtrer et afficher les messages d'erreurs si c'est le cas.
- Installer **networkx** nous permettent de construire le graphe des nœuds visitées et aussi le courbe des données
- Installer le module **matplotlib** pour afficher le graphe des nœuds visitées et aussi la courbe de notre fonction si elle converge ou diverge.

Vous trouvez le code complet sur Git : https://github.com/khaled44000/Algorithmes-de-fourmi.git

II. Installation

Installation via pip3 :

```
pip3 install ACO-Pants
pip3 install csv
pip3 install geopy
pip3 install time
pip3 install pytest-warnings
pip3 install network
pip3 install matplotlib
```

III. Usage

L'utilisation de **Pants** est simple, dans ce TP la distance est entre des nœuds 3D avec des coordonnées (latitude, longitude), c'est pour sa on besoin du la module **geopy** pour calculer la distance entres les nœuds.

1. Importer Pants avec tous les autres modules :

```
# -*- coding: utf-8 -*-
import pants
import csv
import geopy.distance
import time
import networkx as nx
```

```
import warnings
warnings.filterwarnings('ignore')
```

Remarque : il faut bien mettre dans l'entête du code **# -*- coding: utf-8 -*-**

C'est une spécificité de Python : si l'encoding du fichier est différent de l'encoding par défaut du langage, il faut le déclarer sinon le programme plantera à la première conversion.

En Python 2.7, l'encoding par défaut est ASCII, donc il faut presque toujours le déclarer.

En Python 3, l'encoding par défaut est UTF8 et on peut donc l'omettre si on l'utilise.

Ensuite, il existe deux types de chaînes de caractères en Python :

- La chaîne de caractères encodée : type 'str' en Python 2.7, 'byte' en Python 3.
- La chaîne de caractères décodée : type 'unicode' en Python 2.7, et 'str' en python 3 (sic).

2. La méthode **Main()** qui appel les différentes utilisées dans le code

```
def main():
    try:
        readCSV()
        world = createWorld()
        solver = createSolver()
        printSolution(solver, world)
    except Exception as ex:
        print("main : " + format(ex))
```

3. La fonction **readCSV()** nous permet de lire les données dans le fichier CSV et de crées nos points de données ceux-ci deviennent les nœuds. Nous créons des points 3D à partir des latitudes et longitude importées.

```
nodes = []
data = []
fileName = None
tempsdebut = time.time()

def readCSV():
    global fileName
    if fileName == None:
        fileName = input("Entrer le nom du fichier :")  # 1 ou 2 ou 3 ou 4
ou 5
    if(fileName is not  None):
        with open('files/test' + fileName + '.csv', 'r') as csvfile:
            csvReader = csv.reader(csvfile, delimiter=',', quotechar='"')
            next(csvReader)
            for row in csvReader:
                if row:
```

```
                    latitude = row[6]
                    longitude = row[7]

                    print("latitude : " + latitude, "longitude : " +
longitude)
                    try:
                        if (latitude !='' and  longitude != '' and
                                latitude != '/N' and longitude != '/N' and
                                isinstance(float(latitude), float) and
isinstance(float(longitude), float)):
                             nodes.append((float(latitude),
float(longitude)))
                    except Exception as ex:  # latitude contient "/N" ou
longitude contient "/N"
                        print(format(ex))
```

Dans la fonction **readCSV(),** j'ai ajouté un test qui permet de sauter les lignes vides et les lignes qui contiennent des caractères « /N » , si non le programme plantera lors de la calcule des distances surtout.

4. La fonction **remove_duplicates(values)** supprime les doublons dans la liste des nœuds, sinon on rencontre des problème lors de la création de world, comme par exemple des codes erreurs key(183 ,76).

```
def remove_duplicates(values):
    output = []
    seen = set()
    for value in values:
        # Si la valeur n'a pas encore été rencontrée,
        # ... ... ajoutez-le à la liste et à l'ensemble
        if value not in seen:
            output.append(value)
            seen.add(value)
    return output
```

La fonction **remove_duplicates(values),** input : la liste des nœuds qui contient des doublons, la sortie une liste des nœuds propres sans les duplicatas des données.

5. La fonction **CalculDistance(coords_a, coords_b)** est utilisé pour calculer la distance entre deux nœuds (a, b) ; elle prend comme entré une liste de type **TUPLE**
   (coords_a (latitude_a, longitude_a) , **coords_b** (latitude_b, longitude_b))
   Dans cette fonction on utilise le model **gepy** pour calculer la distance 3D entre les nœuds.
   Retourne la distance mesurée en KM

```
def CalculDistance(coords_a, coords_b):
    dist = geopy.distance.vincenty(coords_a, coords_b).km
    # TEST
    # print("dist: " + format(dist)  + '/n')
    return dist
```

6. La fonction **createWorld()** , permet de créer le World , elle fait appel au premier lieu à la fonction **remove_duplicates(values),** pour supprimer les duplicata dans la liste des nœuds , elle prends comme entrée la liste des nœuds avec leurs distances.

```
def createWorld():
    try:
        if(isinstance( nodes, type([]))):
            data = remove_duplicates(nodes)
            return pants.World(data, CalculDistance)
    except Exception as ex:
        print("createWorld : " + format(ex.args))
```

7. La fonction **createSolver(),** permet de créer le solveur grâce à **Pants**

```
def createSolver():
    try:
        return pants.Solver()
    except Exception as ex:
        print("createSolver : " + format(ex))
```

8. La fonction **printSolution(solver, world)** , fait appel à la fonction **methodeSolutions(solver, world)** qui nous renvoie chaque solution trouvée si elle est la meilleure jusqu'à présent, **methodeSolve(solver, world)** renvoie la meilleure solution trouvée et on stock les nœuds visiter dans la liste nœudsVisiter qu'on va utiliser pour construire notre graph dans la fonction **createGraph(neoudsVisiter)**, et la **calculerTempsExecution()** calcule le temps d'exécution du programme. Je vous détaille par la suite chaque fonction appelée.

```
def printSolution(solver, world):
    try:
        if(type(solver) and type(world)):
            # renvoie chaque solution trouvée si elle est la meilleure
jusqu'à présent
            methodeSolutions(solver, world)
            # renvoie la meilleure solution trouvée
            neoudsVisiter = methodeSolve(solver, world)
            # calculer le temps d'exécution
            calculerTempsExecution()
            # affichage du graphe
            createGraph(neoudsVisiter)
    except Exception as ex:
        print("printSolution : " + format(ex))
```

9. La fonction **methodeSolve(solver, world),** renvoie la meilleure distance minimum , elle prend comme entrée le solver et world qui sont déjà été créer, elle affiche la liste des nœuds visité et Edges prises et nous retourne la liste des nœuds visitées

```
def methodeSolve(solver, world):
    try:
        if solver is None or world is None:
            print("solver ou world None")
        else:
            solution = solver.solve(world)  # renvoie la meilleure solution
trouvée
            if solution is None:
                print("solution None")
            # Afficher la Meilleur distance trouvée par solve
            else:
                print("")
                print("-------------------------------------------------------
----------------------------")
                print("Meilleur Distance trouvée par solver.solve(world) :
" + format(solution.distance) + " KM")
                print("-------------------------------------------------------
----------------------------")
                print("Noeuds visités : " + format(solution.tour))  #
noeuds visités
                print("-------------------------------------------------------
----------------------------")
                print("Edges  prises : " + format(solution.path))  # Edges
prises
                print("-------------------------------------------------------
----------------------------")
                neoudsVisiter = solution.tour
                return neoudsVisiter
    except Exception as ex:
        print("error in methodeSolve : " + format(ex))
```

10. La fonction **methodeSolutions(solver, world)** renvoie chaque solution trouvée si elle est la meilleure jusqu'à présent, elle affiche les distances optimaux et choisi d'affiche au final la meilleur distance minimum trouvé.

```
def methodeSolutions(solver, world):
    try:
        # renvoie chaque solution trouvée si elle est la meilleure jusqu'à
présent
        solutions = solver.solutions(world)
        if solutions is None:
            print("solutions None")
        else:
            i = 1
            bestSolution = float("inf")
            for sol in solutions:
                # Afficher les solutions optimaux
                print("")
```

```
                print("Distance " + format(i) + " : " +
format(sol.distance) + " KM")
                i += 1
                assert sol.distance < bestSolution
                bestSolution = sol.distance

                # Afficher la Meilleur distance
            print("")
            print("-----------------------------------------------------------
------------------------")
            print("Meilleur Distance trouvée par solver.solutions(world) :
" + format(bestSolution) + " KM")
            print("-----------------------------------------------------------
------------------------")
            print("")
    except Exception as ex:
        print("error in methodesolutions : " + format(ex))
```

11. La fonction **createGraph(neoudsVisiter),** permet de construire le graph grâce à **networkx** , elle prend la liste les nœuds visitées comme entrées, ces données en les rajoute dans la méthode **add_edges_from (neoudsVisiter)** proposé par **networkx** ou bien la méthode **add_edge().**
    nx.spring_layout(G) : définit les positions de toutes les nœuds
    draw_networkx() : personnaliser les nœuds , définit le couleur des nœuds et le couleur de fil de liaison en les noeuds , la taille du nœuds et z-index.
    draw_networkx_edge_labels() : définit les labels des nœuds ; les label sont les cordonnés des nœuds.
    Une fois j'ai toutes les données sont définit dans les fonctions proposées par le model **networkx,** je fais appel au module **matplotlib** pour afficher le graph des nœuds visitées et le courbe de dégrée séquence qui donne une vision si notre fonction converge ou diverge.

```
def createGraph(neoudsVisiter):
    G = nx.Graph()
    # G.add_edges_from(neoudsVisiter)
    for neoud in neoudsVisiter:
        G.add_edge(format(neoud[0]), format(neoud[1]), weight=0.6)
    plt.subplot(121)

    node_positions = nx.spring_layout(G)  # positions for all nodes
    nx.draw_networkx(G, pos=node_positions, node_size=100,
node_color='red', edge_color="green", with_labels=True,
                     alpha=1)

    edge_labels = nx.get_edge_attributes(G, 'sequence')
    nx.draw_networkx_edge_labels(G, pos=node_positions,
edge_labels=edge_labels, font_size=20)
    nx.draw_networkx_nodes(G, pos=node_positions, node_size=20)
    nx.draw_networkx_edges(G, pos=node_positions, alpha=0.4)
```

```
    plt.xticks([])
    plt.yticks([])

    plt.text(0.5, 0.5, G, ha="center", va="center", size=24, alpha=.5)
    plt.title('Noeuds Visitées', size=15)

    plt.ylabel("Y")
    plt.xlabel("X")
    plt.axis('off')

    plt.subplot(122)
    degree_sequence = sorted([d for n, d in G.degree()], reverse=True)
    print("Degree sequence", degree_sequence)
    dmax = max(degree_sequence)
    plt.loglog(degree_sequence, 'b-', marker='o')
    plt.title("Courbe")
    plt.ylabel("Degree")
    plt.xlabel("Rank")

    # dessine le graphique dans l'encart
    plt.axes([0.45, 0.45, 0.45, 0.45])
    Gcc = sorted(nx.connected_component_subgraphs(G), key=len,
reverse=True)[0]
    pos = nx.spring_layout(Gcc)
    plt.axis('off')
    nx.draw_networkx_nodes(Gcc, pos, node_size=20)
    nx.draw_networkx_edges(Gcc, pos, alpha=0.4)
    plt.xticks([])
    plt.yticks([])
    plt.text(0.5, 0.5, Gcc, ha="center", va="center", size=24, alpha=.5)
    plt.show()
```

12. La fonction **calculerTempsExecution() :** permet de calculer et afficher tout simplement le temps d'exécution du programme.

```
def calculerTempsExecution():
    try:
        # temps fin d'exécution
        tempsfin = time.time()
        # Calculer le temps d'exécution
        TempsEx = tempsfin - tempsdebut
        # Afficher le temps d'exécution
        print("")
        print("-------------------------------------------------------------
--------------------")
        print("Temps d'exécution : " + format(TempsEx) + " secondes")
        print("-------------------------------------------------------------
--------------------")
        print("")
    except Exception as ex:
```

```
        print("calculerTempsExecution : " + format(ex))
```

IV. Test

Ouvrir l'invite de commande (CMD), Aller dans le répertoire là il se trouve le programme.

Taper python Tp.py, ensuite le programme se lancera et vous demande de saisir le nom du fichier CSV

1. Test numéro 1

Dans le test numéro 1 j'ai saisi 1 (le fichier CSV qui contient 20 lignes), le résultat est comme suit :

```
Entrer le nom du fichier :1
latitude : 51.97039 longitude : 0.979328
latitude : 52.094427 longitude : 0.668408
latitude : 52.038683 longitude : 0.730226
latitude : 51.966211 longitude : 0.972091
latitude : 52.102815 longitude : 0.666893
latitude : 51.956771 longitude : 1.268376
latitude : 51.996361 longitude : 1.213605
latitude : 51.978282 longitude : 1.019392
latitude : 51.968265 longitude : 1.088246
latitude : 51.976413 longitude : 1.053112
latitude : 52.077065 longitude : 0.71663
latitude : 51.956682 longitude : 1.057899
latitude : 52.046763 longitude : 0.957949
latitude : 52.09181 longitude : 0.871471
latitude : 52.028751 longitude : 0.743908
latitude : 52.037694 longitude : 0.855736
latitude : 51.968086 longitude : 1.114167
latitude : 52.097693 longitude : 0.667931
latitude : 52.040323 longitude : 0.73088

Distance 1 : 106.74775139377385 KM

Distance 2 : 104.26370668633822 KM

Distance 3 : 103.25258550092049 KM

Distance 4 : 100.67860876957504 KM

--------------------------------------------------------------------------------
Meilleur Distance trouvée par solver.solutions(world) : 100.67860876957504 KM
--------------------------------------------------------------------------------



--------------------------------------------------------------------------------
Meilleur Distance trouvée par solver.solve(world) : 99.96084300784196 KM
--------------------------------------------------------------------------------
```

Figure 1

- Affichage du (latitude, longitude)
- Affichage solutions optimaux et la meilleur distance trouvées par la fonction **solver.solutions(word)**
- Affichage la meilleur distance trouvé par **solver.solve(world)**

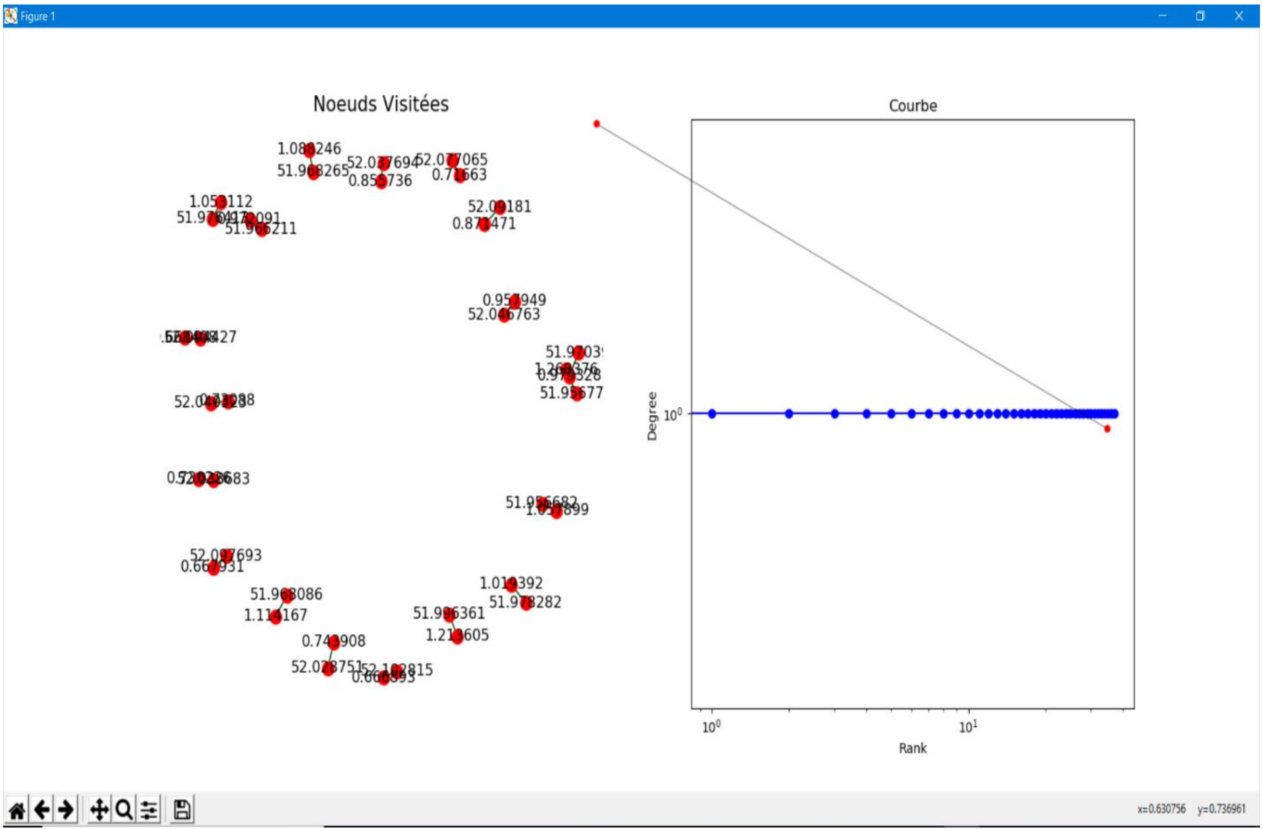
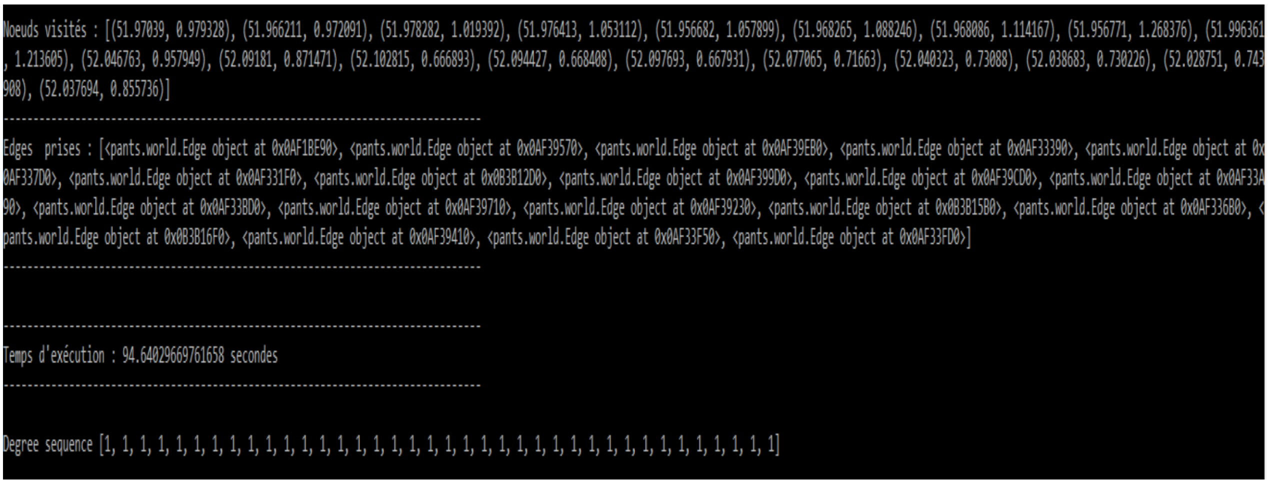


Figure 2

- Affichage du graphe des nœuds visité
- La courbe ne donne pas une vision claire pour l'instant si l'ensemble des nœuds converge ou non



```
Noeuds visités : [(51.97039, 0.979328), (51.966211, 0.972091), (51.978282, 1.019392), (51.976413, 1.053112), (51.956682, 1.057899), (51.968265, 1.088246), (51.968086, 1.114167), (51.956771, 1.268376), (51.996361, 1.213605), (52.046763, 0.957949), (52.09181, 0.871471), (52.102815, 0.666893), (52.094427, 0.668408), (52.097693, 0.667931), (52.077065, 0.71663), (52.040323, 0.73088), (52.038683, 0.730226), (52.028751, 0.743908), (52.037694, 0.855736)]
--------------------------------------------------------------------------------
Edges  prises : [<pants.world.Edge object at 0x0AF1BE90>, <pants.world.Edge object at 0x0AF39570>, <pants.world.Edge object at 0x0AF39EB0>, <pants.world.Edge object at 0x0AF33390>, <pants.world.Edge object at 0x0AF337D0>, <pants.world.Edge object at 0x0AF331F0>, <pants.world.Edge object at 0x0B3B12D0>, <pants.world.Edge object at 0x0AF399D0>, <pants.world.Edge object at 0x0AF39CD0>, <pants.world.Edge object at 0x0AF33A90>, <pants.world.Edge object at 0x0AF33BD0>, <pants.world.Edge object at 0x0AF39710>, <pants.world.Edge object at 0x0AF39230>, <pants.world.Edge object at 0x0B3B15B0>, <pants.world.Edge object at 0x0AF336B0>, <pants.world.Edge object at 0x0B3B16F0>, <pants.world.Edge object at 0x0AF39410>, <pants.world.Edge object at 0x0AF33F50>, <pants.world.Edge object at 0x0AF33FD0>]
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
Temps d'exécution : 94.64029669761658 secondes
--------------------------------------------------------------------------------


Degree sequence [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Figure 3

- Affichage de la liste des nœuds visitées
- Affichage la liste des Edges prises
- Le temps d'exécution 94,64 secondes = 1,57 minutes
- Affiche de la liste Degré de séquence nous permet de savoir si l'ensemble converge ou non, pour l'instant tout l'ensemble est égal à 1.

2. Test numéro 2

Dans le test numéro 2 j'ai saisi 1 (le fichier CSV qui contient 999 lignes), le résultat est comme suit :

```
Distance 1 : 3101.091866709442 KM

Distance 2 : 3098.9109168463165 KM

Distance 3 : 3069.5244140347068 KM

Distance 4 : 3050.4075000523894 KM

Distance 5 : 3043.1700351307145 KM

Distance 6 : 2964.2033826983647 KM

--------------------------------------------------------------------------------
Meilleur Distance trouvée par solver.solutions(world) : 2964.2033826983647 KM
--------------------------------------------------------------------------------



--------------------------------------------------------------------------------
Meilleur Distance trouvée par solver.solve(world) : 2912.8810343175464 KM
--------------------------------------------------------------------------------
```

Figure 1

- Affichage solutions optimaux et la meilleur distance trouvées par la fonction **solver.solutions(word)**
- Affichage la meilleur distance trouvé par **solver.solve(world)**

Figure 2 : liste des nœuds visitées

Figure 3 : suite de la liste des nœuds visités

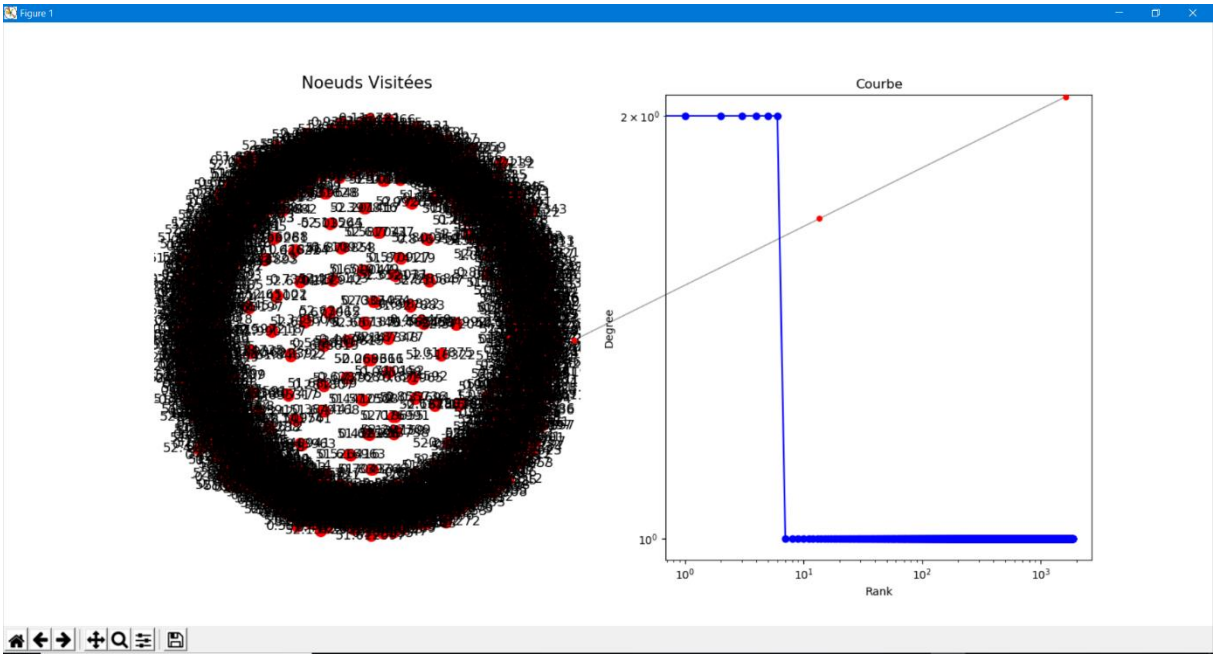


Figure 4

- Affichage du graphe des nœuds visitées
- La courbe nous donne une vision claire de la convergence pour l'ensemble des nœuds

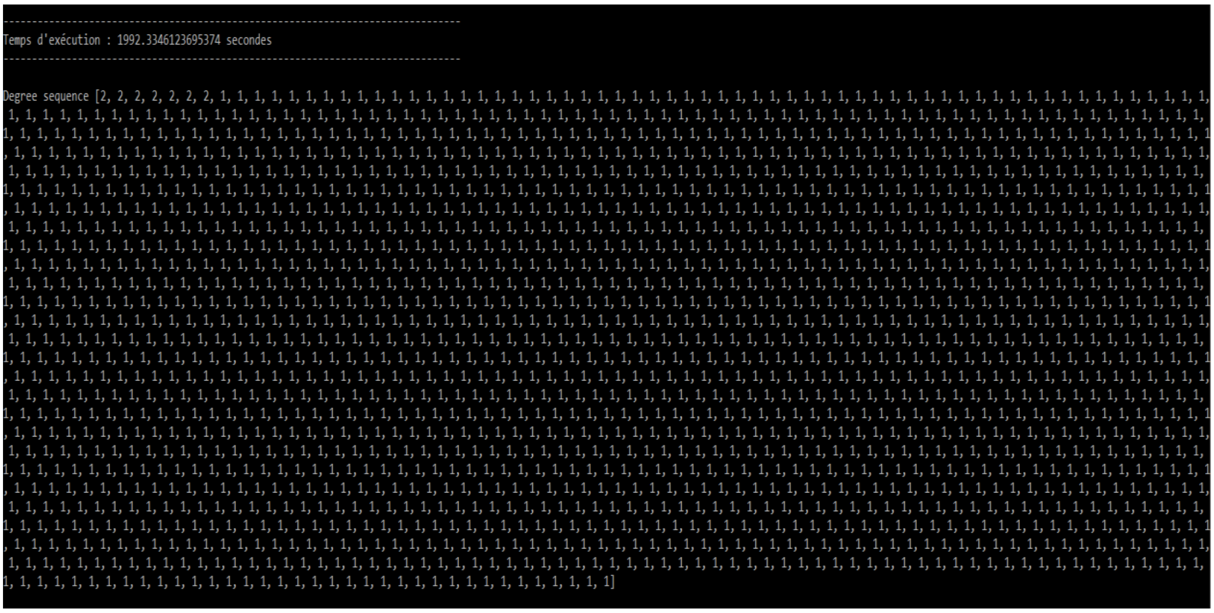


Figure 5

- Le temps d'exécution 1992,33 secondes = 33 minutes
- Affiche de la liste Degré de séquence