كليـــــــة العلوم
السملالية - مراكش
FACULTÉ DES SCIENCES
SEMLALIA - MARRAKECH

# Utilisation des algorithmes TOPSIS et AHP pour l'identification des nœuds influents dans un réseau compliqué.

Master Sciences de données

Année universitaire 2022/2023

Réalisé par :

JALLOULI CHAIMAE

AZNAG ACHIK

CHARAF OUALID

# Table des matières

# Introduction générale

Avec le développement rapide des technologies de l'information, l'échelle des réseaux complexes augmente, ce qui rend la propagation des maladies et des rumeurs plus difficiles à contrôler. L'identification efficace et précise des nœuds influents est essentielle pour prédire et contrôler le système de réseau de manière pertinente.

L'identification des nœuds influents dans un réseau réel est un domaine de recherche actif, vaste et riche d'applications. Marketing viral, propagation de virus, confinement des rumeurs sont parmi les applications les plus connues.

Les nœuds et les bords de différents types de réseaux jouent divers rôles dans la structure et la fonction du réseau. Ces réseaux sont hétérogènes aux échelles macro, méso et micro.

L'un des problèmes les plus difficiles est l'identification des nœuds influents dans les réseaux sociaux dynamiques qui a attiré une attention croissante ces dernières années. Les vrais réseaux sociaux comme Facebook ou Twitter sont très changeants donc ils sont représentés comme des graphiques temporels qui évoluent dans le temps.

Les mesures de centralité sont des méthodes bien connues utilisées pour quantifier l'influence des nœuds en extrayant des informations de la structure du réseau. L'écueil de ces mesures est de repérer des nœuds situés à proximité les uns des autres, saturant leur zone d'influence commune. Néanmoins, ces mesures restent limitées et donnent des résultats moins performants par rapport aux autres algorithmes plus avancé.

Dans ce projet, nous allons entamer cette identification en utilisant des algorithmes notamment TOPSIS, WTOPSIS et AHP. Tout en introduisant les méthodes et outils implémentés, présentant les différentes étapes suivies et démontrant la mise en œuvre du projet.

# Chapitre I. Contexte général

## Introduction

Dans ce chapitre, nous allons effectuer une étude théorique, qui va servir comme base nécessaire pour comprendre le reste du projet. Cette étude théorique concerne la présentation des réseaux complexes, l'utilisation de la théorie des graphes et finalement la détection des nœuds influents.

## Réseaux complexes

Dans le contexte de la théorie des réseaux, un réseau complexe est un graphe (réseau) avec des caractéristiques qui ne se produisent pas dans des réseaux simples tels que des treillis ou des graphes aléatoires mais se produisent souvent dans des réseaux représentant des systèmes réels.

L'étude des réseaux complexes est un domaine de recherche scientifique jeune et actif (depuis 2000) inspiré en grande partie par les découvertes empiriques de réseaux du monde réel tels que les réseaux informatiques, les réseaux biologiques, les réseaux technologiques, les réseaux cérébraux, réseaux climatiques et réseaux sociaux.

## Graphes : outils de modélisation

Un graphe est une collection d'éléments mis en relation entre eux. Géométriquement, on représente ces éléments par des points (les sommets) reliés entre eux par des arcs de courbe (les arêtes). Selon que l'on choisit d'orienter les arêtes ou de leur attribuer un poids (un coût de passage), on parle de graphes orientés ou de graphes pondérés.

La théorie des graphes s'intéresse à leurs multiples propriétés : existence de chemins les plus cours, de cycles particuliers, nombre d'intersections dans le plan, problèmes de coloriage…

Aujourd'hui, les graphes trouvent plusieurs applications dans la modélisation des réseaux (routiers, informatiques, etc..). Par ailleurs, la théorie des graphes a fourni des problèmes algorithmiques cruciaux en théorie de la complexité.

## Détection des nœuds influents

Dans les réseaux, tous les nœuds n'ont pas la même importance, et certains sont plus importants que d'autres. La question de trouver les nœuds les plus importants dans les réseaux a été largement abordée, en particulier pour les nœuds dont l'importance est liée à la connectivité du réseau.

Ces nœuds sont généralement appelés nœuds critiques. Le problème de détection de nœud critique (CNDP) est le problème d'optimisation qui consiste à trouver l'ensemble de nœuds dont la suppression dégrade au maximum la connectivité du réseau selon certaines métriques de connectivité prédéfinies. Prédéfinie, différentes variantes ont été développées.

# Chapitre II. Outils de détection des nœuds influents

Dans ce chapitre, nous allons présenter les différentes méthodes de détection des nœuds influents accompagnés des algorithmes d'évaluation, passant du plus simple au plus avancé.

## 1. Méthodes de détection des nœuds influents

***Mesures de centralité***

Les mesures de centralité sont un outil essentiel pour comprendre les réseaux, souvent aussi appelés graphes.

Ces algorithmes utilisent la théorie des graphes pour calculer l'importance d'un nœud donné dans un réseau. Ils coupent les données bruyantes, révélant les parties du réseau qui nécessitent une attention, mais ils fonctionnent tous différemment. Chaque mesure a sa propre définition de "l'importance", vous devez donc comprendre leur fonctionnement pour trouver la meilleure pour vos applications de visualisation de graphiques.

- Degree Centrality : Attribue un score d'importance basé simplement sur le nombre de liens détenus par chaque nœud. Cette mesure nous montre le nombre de connexions directes chaque nœud a-t-il avec d'autres nœuds du réseau. Elle est souvent utilisée pour trouver des personnes très connectées, des personnes populaires, des personnes susceptibles de détenir la plupart des informations ou des personnes pouvant se connecter rapidement au réseau plus large.

- Betweenness Centrality : Mesure le nombre de fois qu'un nœud se trouve sur le chemin le plus court entre d'autres nœuds. Cette mesure montre quels nœuds sont des « ponts » entre les nœuds d'un réseau. Pour ce faire, il identifie tous les chemins les plus courts, puis compte le nombre de fois où chaque nœud tombe sur un. Cette mesure est utilisée Pour trouver les individus qui influencent le flux autour d'un système.

- Closeness Centrality : Note chaque nœud en fonction de sa « proximité » avec tous les autres nœuds du réseau. Cette mesure calcule les chemins les plus courts entre tous les nœuds, puis attribue à chaque nœud un score basé sur sa somme des chemins les plus courts, et elle est utilisée pour trouver les individus les mieux placés pour influencer le plus rapidement l'ensemble du réseau.

- Eigenvector Centrality : Comme Degree Centrality, EigenCentrality mesure l'influence d'un nœud en fonction du nombre de liens qu'il a avec d'autres nœuds du réseau. EigenCentrality va ensuite un peu plus loin en prenant également en compte le niveau de connexion d'un nœud, le nombre de liens de ses connexions, etc. à travers le réseau. Cette mesure calcule les connexions étendues d'un nœud, EigenCentrality peut identifier les nœuds ayant une influence sur l'ensemble du réseau, pas seulement ceux qui y sont directement

connectés. Elle est pratique pour comprendre les réseaux sociaux humains, mais aussi pour comprendre les réseaux comme la propagation des logiciels malveillants.

***L'algorithme TOPSIS***

TOPSIS, connue sous le nom de Technique for Order of Preference by Similarity to Ideal Solution, est une méthode d'analyse décisionnelle multicritères. Il compare un ensemble d'alternatives sur la base d'un critère prédéfini. La méthode est utilisée dans l'entreprise dans diverses industries, chaque fois que nous devons prendre une décision analytique basée sur les données collectées.

La logique de TOPSIS est basée sur le concept que l'alternative choisie doit avoir la distance géométrique la plus courte de la meilleure solution et la distance géométrique la plus longue de la pire solution.

Une telle méthodologie permet de trouver des compromis entre les critères lorsqu'une mauvaise performance sur l'un peut être annulée par une bonne performance sur un autre critère. Cela fournit une forme de modélisation assez complète car nous n'excluons pas des solutions alternatives basées sur des seuils prédéfinis.

- Etapes de l'algorithme Topais

i- Création d'une matrice d'évaluation de M alternatives et N critères

$$(a_{ij})_{MxN}$$

Dans notre exemple, les alternatives seront les différents nœuds du réseau, et les critères seront les mesures de centralité.

ii- Normalisation de la matrice d'évaluation

$$\alpha_{ij} = \frac{a_{ij}}{\sqrt{\sum_{i=1}^{M}(a_{ij})^2}}$$

iii- Calcul du Weighted Normalized Matrix

$$\chi_{ij} = \alpha_{ij} * \omega_j$$

$$\sum_{j=1}^{N} \omega_j = 1$$

iv- Déterminer le meilleur et le pire alternative pour chaque critère

$$\chi_j^b = max\, \chi_{ij}$$

$$\chi_j^w = min\, \chi_{ij}$$

v- Calculer la distance euclidienne entre l'alternative et meilleur/pire alternative

$$d_j^b = \sqrt{\sum_{j=1}^{N} (\chi_j^i - \chi_j^b)^2}$$

$$d_j^w = \sqrt{\sum_{j=1}^{N} (\chi_j^i - \chi_j^w)^2}$$

vi- Calculer la similarité entre chaque alternative et le pire alternative

$$S_i = \frac{d_i^w}{d_i^w + d_i^b}$$

vii- Trier les alternatives selon la valeur du Topsis dans l'ordre décroissant.

De cette manière, on obtient un ensemble d'alternatives ordonnés selon des critères spécifiques.

## 2. Méthodes comparatives

***Modèle SI (Susceptible/Infected)***

Le modèle le plus simple d'un nœud infectieux catégorise les nœuds comme susceptibles ou infectieux (SI). On peut imaginer que les nœuds sensibles sont en bonne santé et que les nœuds infectieux sont malades.

Un nœud sensible peut devenir contagieux par contact avec un infectieux. Ici, et dans tous les modèles ultérieurs, nous supposons que la population étudiée est bien mélangée, de sorte que chaque nœud a une probabilité égale d'entrer en contact avec tous les autres nœuds.

# Chapitre III. Implémentation des algorithmes TOPSIS et w-TOPSIS pour la détection des nœuds influents

## Introduction

Dans cette partie, on attaque la partie pratique du projet. Comme première étape, nous avons implémenté l'algorithme TOPSIS accompagné des 4 mesures de centralité : Degree Centrality, Betweenness Centrality, Closeness Centrality et Eigenvector Centrality.

## 1. Ensemble de données

Pour ce projet, nous avons choisi d'utiliser le dataset « Ego-Facebook ». Ce réseau dirigé contient 2900 nœuds qui représentent des amitiés utilisateur-utilisateur de Facebook. Un nœud représente un utilisateur. Un bord indique que l'utilisateur représenté par le nœud de gauche est un ami de l'utilisateur représenté par le nœud de droite.

## 2. Outils techniques

- Langage de programmation : Python
- Libraires implémentées

  - Numpy
  - Pandas
  - MatplotLib
  - NetworkX
  - Math
  - …

## 3. Implémentation de l'algorithme TOPSIS

Dans cette partie, on présente les segments du code trouvé au niveau du notebook attaché à ce rapport avec la description de chaque étape.

  - Importation des libraires nécessaires

```python
import networkx as nx
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from math import *
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
```

- Lecture du graphe d'après le fichier texte téléchargé avec le dataset « Ego-Facebook » et affichage de ses informations

```python
g = nx.read_edgelist('facebook_combined.txt',create_using=nx.Graph(),nodetype=int)

# check of data has been loaded correctly

info=nx.info(g)
print(info)
```

- Pour l'implémentation de l'algorithme TOPSIS, on a utilisé une classe Topsis(), qui a les attributs dont on aura besoin (Nombre de ligne, Nombre de colonnes, La matrice d'évaluation…)

```python
class Topsis():
    evaluation_matrix = np.array([])  # Matrix
    weighted_normalized = np.array([])  # Weight matrix
    normalized_decision = np.array([])  # Normalisation matrix
    M = 0  # Number of rows
    N = 0  # Number of columns

    '''
```

- Le constructeur de la classe est comme ceci :

```python
def __init__(self, evaluation_matrix, weight_matrix, criteria):
    # MxN matrix
    self.evaluation_matrix = np.array(evaluation_matrix, dtype="float")

    # M alternatives (options)
    self.row_size = len(self.evaluation_matrix)

    # N attributes/criteria
    self.column_size = len(self.evaluation_matrix[0])

    # N size weight matrix
    self.weight_matrix = np.array(weight_matrix, dtype="float")
    self.weight_matrix = self.weight_matrix/sum(self.weight_matrix)
    self.criteria = np.array(criteria, dtype="float")
```

- La deuxième étape consiste à calculer la matrice d'évaluation normalisée :

```
def step_2(self):
    # normalized scores
    self.normalized_decision = np.copy(self.evaluation_matrix)
    sqrd_sum = np.zeros(self.column_size)
    for i in range(self.row_size):
        for j in range(self.column_size):
            sqrd_sum[j] += self.evaluation_matrix[i, j]**2
    for i in range(self.row_size):
        for j in range(self.column_size):
            self.normalized_decision[i,
                                     j] = self.evaluation_matrix[i, j]/(sqrd_sum[j]**0.5)
```

- La troisième étape consiste à calculer la « Weighted Matrix » :

```
def step_3(self):
    from pdb import set_trace
    self.weighted_normalized = np.copy(self.normalized_decision)
    for i in range(self.row_size):
        for j in range(self.column_size):
            self.weighted_normalized[i, j] *= self.weight_matrix[j]
```

- Calculer les meilleurs et pires alternatives :

```
def step_4(self):
    self.worst_alternatives = np.zeros(self.column_size)
    self.best_alternatives = np.zeros(self.column_size)
    for i in range(self.column_size):
        if self.criteria[i]:
            self.worst_alternatives[i] = min(
                self.weighted_normalized[:, i])
            self.best_alternatives[i] = max(self.weighted_normalized[:, i])
        else:
            self.worst_alternatives[i] = max(
                self.weighted_normalized[:, i])
            self.best_alternatives[i] = min(self.weighted_normalized[:, i])
```

- La dernière étape affiche le résultat de chaque étape séparée :

```
def calc(self):
    print("Step 1\n", self.evaluation_matrix, end="\n\n")
    self.step_2()
    print("Step 2\n", self.normalized_decision, end="\n\n")
    self.step_3()
    print("Step 3\n", self.weighted_normalized, end="\n\n")
    self.step_4()
    print("Step 4\n", self.worst_alternatives,
          self.best_alternatives, end="\n\n")
    self.step_5()
    print("Step 5\n", self.worst_distance, self.best_distance, end="\n\n")
    self.step_6()
    print("Step 6\n", self.worst_similarity,
          self.best_similarity, end="\n\n")
```

- Par la suite, on calcule les différentes mesures de centralité, à l'utilisation de la librairie NetworkX

```
[ ]  dc=nx.degree_centrality(g)
     bc=nx.betweenness_centrality(g)
     cc=nx.closeness_centrality(g)
     ec=nx.eigenvector_centrality(g)
```

- Le résultat donnera un dictionnaire, qu'on trie et stocke au niveau d'une liste pour générer un DataFrame contenant la matrice d'évaluation :

| | node | dc | bc | cc | ec |
|---|---|---|---|---|---|
| 0 | 0 | 0.085934 | 1.463059e-01 | 0.353343 | 3.391796e-05 |
| 1 | 1 | 0.004210 | 2.783274e-06 | 0.261376 | 6.045346e-07 |
| 2 | 2 | 0.002476 | 7.595021e-08 | 0.261258 | 2.233461e-07 |
| 3 | 3 | 0.004210 | 1.685066e-06 | 0.261376 | 6.635648e-07 |
| 4 | 4 | 0.002476 | 1.840332e-07 | 0.261258 | 2.236416e-07 |
| ... | ... | ... | ... | ... | ... |
| 4034 | 4034 | 0.000495 | 0.000000e+00 | 0.183989 | 2.951270e-10 |
| 4035 | 4035 | 0.000248 | 0.000000e+00 | 0.183980 | 2.912901e-10 |
| 4036 | 4036 | 0.000495 | 0.000000e+00 | 0.183989 | 2.931223e-10 |
| 4037 | 4037 | 0.000991 | 7.156847e-08 | 0.184005 | 2.989233e-10 |
| 4038 | 4038 | 0.002229 | 6.338922e-07 | 0.184047 | 8.915175e-10 |

[4039 rows x 5 columns]

- On applique l'algorithme Topsis sur cette matrice, on la donnant comme argument, ainsi que la liste des poids [0.2,0.3,0.2,0.3] qui a donné un résultat optimal. On appelle la fonction Topsis.calc() pour afficher les résultats de chaque étape :

```
t = Topsis(evaluation_matrix=ev_matrix_np, weight_matrix=[0.2,0.3,0.3,0.2], criteria=[True,True,True,True])
t.calc()
```

```
Step 1
 [[8.59336305e-02 1.46305921e-01 3.53342667e-01 3.39179617e-05]
 [4.21000495e-03 2.78327442e-06 2.61376141e-01 6.04534613e-07]
 [2.47647350e-03 7.59502118e-08 2.61257764e-01 2.23346094e-07]
 ...
 [4.95294700e-04 0.00000000e+00 1.83988700e-01 2.93122343e-10]
 [9.90589401e-04 7.15684688e-08 1.84005468e-01 2.98923251e-10]
 [2.22882615e-03 6.33892152e-07 1.84047402e-01 8.91517473e-10]]
```

- Le résultat de la dernière étape :

```
Step 4
 [4.61189980e-05 0.00000000e+00 3.02112641e-03 1.27705307e-14] [0.04819435 0.19446755 0.00779116 0.01908139]

Step 5
 [0.06139487 0.00159033 0.00146673 ... 0.00010757 0.00016924 0.00038179] [0.14034886 0.20109779 0.20117525 ... 0.20128929 0.20126723 0.20121211]

Step 6
 [0.30432108 0.0078462  0.00723802 ... 0.00053413 0.00084018 0.00189387] [0.69567892 0.9921538  0.99276198 ... 0.99946587 0.99915982 0.99810613]
```

- A ce niveau, on passe à calculer Closeness, on utilisant les meilleurs et pires alternatives :

```
best_dist = np.array(t.best_distance)
worst_dist = np.array(t.worst_distance)
closenness = []
closenness = worst_dist / (worst_dist + best_dist)

C_df=pd.DataFrame(closenness,columns=['C'])

print(closenness)
```

```
[0.30432108 0.0078462  0.00723802 ... 0.00053413 0.00084018 0.00189387]
```

- On génère un DataFrame contenant la meilleur et la pire distance de chaque nœud et la valeur du Closeness obtenue :

```
closeness_topsis = pd.DataFrame(
    {
        'S+': best_dist,
        'S-': worst_dist,
        'C': closenness,
        'node2':nodes_list
    }
)

closeness_topsis
```

| | S+ | S- | C | node2 |
|---|---|---|---|---|
| 0 | 0.140349 | 0.061395 | 0.304321 | 107 |
| 1 | 0.201098 | 0.001590 | 0.007846 | 1684 |
| 2 | 0.201175 | 0.001467 | 0.007238 | 1912 |
| 3 | 0.201098 | 0.001590 | 0.007846 | 3437 |
| 4 | 0.201175 | 0.001467 | 0.007238 | 0 |
| ... | ... | ... | ... | ... |

- Pour les 10 premiers :

```
[ ] closeness_topsis.head(10)
```

| | S+ | S- | C | node2 |
|---|---|---|---|---|
| 0 | 0.140349 | 0.061395 | 0.304321 | 107 |
| 1 | 0.201098 | 0.001590 | 0.007846 | 1684 |
| 2 | 0.201175 | 0.001467 | 0.007238 | 1912 |
| 3 | 0.201098 | 0.001590 | 0.007846 | 3437 |
| 4 | 0.201175 | 0.001467 | 0.007238 | 0 |
| 5 | 0.201142 | 0.001513 | 0.007464 | 1085 |
| 6 | 0.201219 | 0.001424 | 0.007029 | 698 |
| 7 | 0.200996 | 0.001857 | 0.009154 | 567 |
| 8 | 0.201197 | 0.001443 | 0.007120 | 58 |
| 9 | 0.200665 | 0.002947 | 0.014476 | 428 |

- On concatène les DataFrames générés :

```
[ ] result = pd.concat([closeness_topsis, centralities_closeness], axis=1)
    result.head(10)
```

| | S+ | S- | C | node2 | node | dc | bc | cc | ec | C |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.140349 | 0.061395 | 0.304321 | 107 | 107 | 0.258791 | 0.480518 | 0.459699 | 2.606940e-04 | 0.913277 |
| 1 | 0.201098 | 0.001590 | 0.007846 | 1684 | 1684 | 0.196137 | 0.337797 | 0.393606 | 7.164260e-06 | 0.695566 |
| 2 | 0.201175 | 0.001467 | 0.007238 | 1912 | 1912 | 0.186974 | 0.229295 | 0.350947 | 9.540696e-02 | 0.496063 |
| 3 | 0.201098 | 0.001590 | 0.007846 | 3437 | 3437 | 0.135463 | 0.236115 | 0.314413 | 9.531613e-08 | 0.488865 |
| 4 | 0.201175 | 0.001467 | 0.007238 | 0 | 0 | 0.085934 | 0.146306 | 0.353343 | 3.391796e-05 | 0.304321 |
| 5 | 0.201142 | 0.001513 | 0.007464 | 1085 | 1085 | 0.016345 | 0.149015 | 0.357852 | 3.164082e-06 | 0.297379 |
| 6 | 0.201219 | 0.001424 | 0.007029 | 698 | 698 | 0.016840 | 0.115330 | 0.271189 | 1.116876e-09 | 0.231106 |
| 7 | 0.200996 | 0.001857 | 0.009154 | 567 | 567 | 0.015602 | 0.096310 | 0.328881 | 9.932295e-06 | 0.193646 |
| 8 | 0.201197 | 0.001443 | 0.007120 | 58 | 58 | 0.002972 | 0.084360 | 0.397402 | 5.898120e-04 | 0.169464 |
| 9 | 0.200665 | 0.002947 | 0.014476 | 428 | 428 | 0.028479 | 0.064309 | 0.394837 | 5.990065e-04 | 0.132923 |

## Utilisation du modèle comparatif SI

- On effectue quelques modifications sur notre résultat pour l'adapter au format désiré, pour appliquer le modèle SI :

```
def SI(node):
    n = nx.number_of_nodes(g)
    model = ep.SIModel(g)
    cfg = mc.Configuration()
    cfg.add_model_parameter('beta', 1)
    cfg.add_model_initial_configuration('Infected', node)
    model.set_initial_status(cfg)
    res = pd.DataFrame(columns=['iteration', 'nb_Susceptible', 'Nb_infected'])
    for i in range(n):
        iteration = model.iteration()
        res.loc[len(res.index)] = [iteration['iteration'], iteration['node_count'][0], iteration['node_count'][1] ]
        if iteration['node_count'][1] == n:
            break
    return res
```

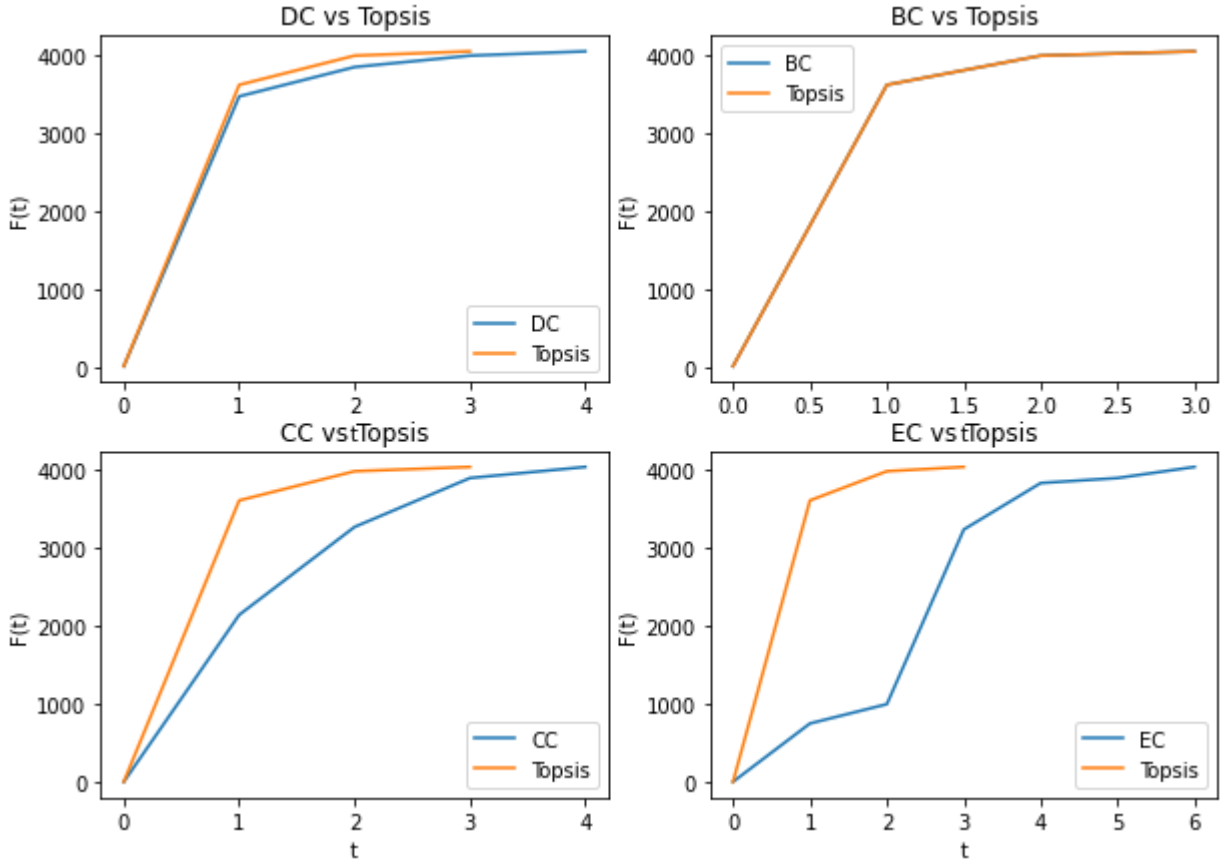
- Cet algorithme est appliqué aux 10 premières valeurs du DataFrame, pour chaque colonne individuellement

```
Rank_DC = SI(set(topten['dcn']))
Rank_BC = SI(set(topten['bcn']))
Rank_CC = SI(set(topten['ccn']))
Rank_EC = SI(set(topten['ecn']))
Rank_Topsis = SI(set(topten['node2']))

Rank_DC
```

| | iteration | nb_Susceptible | Nb_infected |
|---|---|---|---|
| 0 | 0 | 4029 | 10 |
| 1 | 1 | 576 | 3463 |
| 2 | 2 | 201 | 3838 |
| 3 | 3 | 55 | 3984 |
| 4 | 4 | 0 | 4039 |

- De cette manière, on est arrivé à visualiser les courbes




## Implémentation de l'algorithme w-TOPSIS

Pour appliquer cet algorithme, nous avons principalement passé par les mêmes étapes de l'algorithme TOPSIS régulier, c'est-à-dire :

- L'acquisition et génération des données
- Normalisation de la donnée
- Détermination du poids
- Sélection des meilleurs alternatives en utilisant TOPSIS

Au niveau de l'algorithme w-TOPSIS, une étape supplémentaire s'ajoute afin de calculer des poids à valeurs plus optimales :

- On calcule la valeur de l'entropie relativement aux éléments de la matrice d'évaluation en effectuant des calculs purement mathématiques :

```
#ev_matrix_np
#pij = xij/sum(xij)
m = nx.number_of_nodes(g)

sum = np.zeros(4)

for i in range(m):
  for j in range(4):
    sum[j] += ev_matrix_np[i][j]

p = np.copy(ev_matrix_np)

for i in range(m):
  for j in range(4):
    p[i][j] = ev_matrix_np[i][j]/sum[j]
```

```
for j in range(4):
  temp_sum = 0
  for i in range(m):
    try:
       pij = p[i][j]
       temp_sum += pij*log(pij)
    except:
      pij = 1
      temp_sum += pij*log(pij)
  E.append( -k*temp_sum)

E
```

- On utilise les valeurs d'entropie pour calculer les poids :

```
D = []
for i in range(4):
  D.append(1-E[i])
D
```

```
[0.061278794514093016,
 0.5931902858544127,
 0.001066783119935133,
 0.31419210410983556]
```

- Les poids trouvés sont :

```
[ ]  sm = np.sum(D)
     weights = [D[i]/sm for i in range(4)]
     weights
```

```
[0.06319173681858646,
 0.6117079280734432,
 0.0011000849264740018,
 0.32400025018149636]
```

- On procède par les mêmes étapes précédentes