Algorithmique de graphe - IN403

Tables de routage

Zeyneb Bouabdallaoui, Ines Lebib, Mohamed Abbas, Nicolas Fond

N° : 21904931/21913521/21914579/21908626 May 10, 2021

Introduction

Le projet consiste à créer une table de routage à partir d'un graphe généré "aléatoirement" contenant 100 sommets. Nous avons décidé de créer ce projet en *Python* pour faciliter son développement et sa compréhension. Nous allons ici détailler les structures de données ainsi que les algorithmes implémentés.

1 Implémentation des graphes

Nous allons décrire dans cette partie une manière d'implémenter les graphes en Python, de la création du graphe à l'ajout de sommet.

Les graphes sont assez simple à implémenter en *Python*, il suffit de faire appel à un *type construit* : les dictionnaires. Nous avons aussi besoin du poids des arêtes. Pour ça il suffit de créer des *dictionnaire imbriqués* (*nested dictionary* en anglais). Ils se présentent sous la forme suivante:

```
>>> Graph = {
    'A':{'B': 34,
         'C': 23},

    'B':{'A': 34},

    'C':{'A': 23}
  }
```

Il y a plusieurs avantages à utiliser des dictionnaires:

- Accès au poids d'une arête (ici entre A et B dans un graphe G) : `G['A']['B']`
- Ajouter des arêtes entre un sommet existant ou non (ici on ajoute à A un nouveau voisin C avec une arête de poids 5): `G['A']['C'] = 5`
- Avoir la liste des voisins d'un sommet: `G['A']`
- etc...

Pour rendre le code principal plus facile à lire, nous avons créé deux structures : `Vertex` et `Graph` qui contiendront réspéctivement les informations d'un sommet (nom et voisin(s)) et les sommets du graphe. Un `Vertex` (sommet en anglais) est un dictionnaire contenant tous les sommets voisins du sommet en question. Par exemple `A = {'B':34, 'C':23}` signifie que le sommet A contient B et C en voisins et qu'ils sont reliés respectivement par un poids de 34 et 23. Le problème étant que pour le graphe, il est plus facile d'avoir des caractères pour désigner un sommet qu'une variable. C'est pourquoi notre structure `vertex` contient aussi le nom du sommet.

```
class Vertex:
    def __init__(self, vertex):
        self.name = vertex
        self.neighbors = {}
```

Notre structure graphe contient, comme dit précédement, tous les `vertices` (sommets) créer en amont, concaténés sous forme d'un dictionnaire.

```
class Graph:
    def __init__(self):
        self.vertices = {}
```

1.1 Création de sommets

La création de sommets se fait à partir d'un nom donné à la fonction: `sommet1 = Vertex('A')`. On a ici créé un sommet `sommet1` du nom de A. On peut faire appel au nom en faisant `sommet1.name`. Pour ajouter des voisins au sommet, nous avons implémenté une méthode `add_neighbor(self, neighbor, weight)` qui prend en argument un voisin de type vertex et un poids de type entier. Pour ajouter un sommet B (créé au préalable) comme voisin à A avec une arête de poids 3 on fera donc `A.add_neighbor(B,3)`.

À noter que nous sommes ici dans un graphe non orienté. Si on ajoute un sommet `'A' :{'B' : 3}` il faut obligatoirement qu'il y est `'B' :{'A' : 3}`. C'est ce que fait cette fonction d'ajout d'un voisin.

1.2 Création du graphe

Notre class 'Graph' comprend simplement un dictionnaire de sommets comme vu plus haut. Ce sont les fonctions de la structure qui sont importantes ici. En effet, on créer les fonctions "outils" qui vont nous permettre de programmer le corps du projet. Un graphe se crée en faisant `G = Graph()` (G ne contient alors aucun sommet). Nous avons donc fait une méthode `add_vertex(self,vertex)` qui prend en argument une variable de type vertex. Si on veux par exemple ajouter notre sommet A au graphe, on fera `G.add_vertex(A)`.

Un autre méthode importante pour la suite est l'ajout d'arête entre deux sommets existant du graphe. Notre fonction `add_edge(self, vertex1_name, vertex2_name, weight)` prend en argument deux noms de vertex (des chaînes de caractères), et un poids de type entier. Pour ajouter une arête de poids 8 entre un sommet qui s'appelle A et un autre qui s'appelle D (tout deux déjà dans le graphe) on entrera : `G.add_edge('A', 'B', 8)`.

Maintenant que nous avons tous les outils en main, nous pouvons commencer le corps du programme. Toutes les fonctions et méthodes n'ont évidement pas été décrites ici, nous le ferons quand nous aurons besoin de celles-ci.

2 Génération du graphe

2.1 Tier 1 - Backbone

La première fonction à implémenter est celle qui consiste à la création des sommets *backbones*. Plusieurs contraintes nous sont ici présentées:

- création de 10 nœuds connectés entre eux,
- 75% de chance qu'une liaison existe entre deux sommets backbones,
- Si la liaison existe, elle doit être évaluée par un poids compris entre 5 et 10.

Notre fonction créé tout d'abord les 10 nœuds et leur donne un nom "$T1\_i$" où i est le $i^{ème}$ tier 1 créé. Pour l'instanciation d'arête, l'idée est de parcourir tous les sommets du graphe et de l'associer à chaque autre sommet de tier 1 s'il est tombé sur les 75%. On a donc l'algorithme suivant:

Algorithme 1 : Création des backbones

Entrées : Graphe G
Résultat : Graphe G avec les sommets de tier 1
pour i *allant de 1 à 10 compris* **faire**
    Ajouter à G le sommet ($T1\_i$);   `// ` i `=` $i^{ème}$ `itération de la boucle`
fin
`/* Variable nous permettant de ne pas repasser sur un sommet déjà traité */`
$G\_tmp = G$
pour chaque *sommet* S *de* G **faire**
    pour chaque *sommet* S_tmp *de* G_tmp **faire**
        si *(valeur Aléatoire entre 0 et 100)* $<$ *75 et* $S! = S\_tmp$ **alors**
            Ajouter une arête à G entre S et S_tmp avec une valeur entre 5 et 10
        fin
    fin
    $G\_tmp = G\_tmp$ privé de S;   `// On enlève le sommet traité de` G_tmp
fin
Retourner G

En Python, la valeur aléatoire est créée avec la librairie `random`. Une valeur entière alétaoire entre 0 et 100 se fait avec `random.randrange(100)`.

2.2 Tier 2 - opérateurs de transit

Les opérateurs de transit (Tier 2) ont plusieurs contraintes qui nécessiteront de faire d'autres fonctions dans notre classe `graph`. Tout d'abord, les contraintes sont :

- création de 20 noeuds,
- connexions :
  - à 1 ou 2 noeud(s) de tier 1,
  - à 2 ou 3 noeuds de tier 2.
- les liaisons seront évaluées entre 10 et 20.

De même que pour le tier 1, les sommets de tier 2 sont tout d'abord créés sans voisins grâce à une boucle qui va cette fois ci faire 20 tours. A chaque itération, on viendra ajouter ces sommets au graphe pris en paramètre de la fonction.

On fait ensuite une boucle qui va entraîner un lien entre un sommet de niveau 1 et un sommet de niveau 2. Pour ça on parcours tous les sommets de tier 2 et on refait une boucle qui ira de 0 à un entier pris au hasard entre 1 et 2. À chaque itération on ajoutera une liaison entre le sommet actuel de la boucle de parcours et un sommet pris au hasard.

La création de lien entre deux sommets de tier 2 nécessite de vérifier plusieurs éléments :

- Si le lien entre ces deux sommets T2 n'existe pas déjà,
- Si le sommet T2 à lier ou le sommet T2 pris au hasard n'est pas déjà connecté à trois autres sommets de tier 2,
- Si les 2 sommets ne sont pas égaux.

On peut voir ci dessous l'algorithme correspondant (algorithme 2).

Comme on peut le voir dans l'algorithme, plusieurs fonctions sont à prévoir du côté du programme Python. La première est celle qui prend un sommet aléatoire. Comme on aura plusieurs fois besoin de cette fonction, on va la rendre plus "flexible" en lui mettant en paramètre le tier à prendre au hasard. Par exemple, `G.pick_random_vertex('T2')` prendra un sommet aléatoire parmi les sommets de tier 2 du graphe. Pour ça, il existe en python une fonction qui permet de prendre aléatoirement un élément dans une liste. On va donc mettre tous les sommets dans une liste puis appeler cette fonction en faisant `vertex\_name = random.choice(list_tier_choisi)` (avec `list\_tier` la liste des sommets de tier donnée en argument de la fonction) et retourner le nom du sommet choisi.

Une autre fonction utile ici est celle qui compte le nombre de voisins d'un tier donné. Cette fonction va permettre de tester si le sommet n'est pas déjà connecté à trois autres sommets de tier 2. L'implémentation de cette fonction en python est plutôt facile. Les paramètres de

Algorithme 2 : Création des opérateurs de transit

Entrées : Graphe G contenant les sommets de tier 1
Résultat : Graphe G avec les sommets de tier 2
pour *i allant de 1 à 20 compris* **faire**
  Ajouter à G le sommet $(T2\_i)$; `// ` i ` = ` $i^{ème}$ ` itération de la boucle`
fin
pour chaque *sommet S de tier 2 du graphe G* **faire**
  pour *y allant de 0 à (valeurAléatoire entre 1 et 2)* **faire**
    Ajouter une arête entre un sommet de tier 1 pris au hasard et le sommet S
  fin
fin
pour chaque *sommet S de tier 2 du graphe G* **faire**
  pour *k allant de 0 à (valeurAléatoire entre 2 et 3)* **faire**
    $S\_rand =$ Sommet de tier 2 pris au hasard
    si *le sommet S est conneté à strictement moins de 3 sommets de tier 2* **alors**
      tant que *le sommet S est égal au sommet S_rand*
      OU S_rand est déjà connecté à 3 autres tier 2
      OU que S est déjà lié à S_rand **faire**
        $S\_rand =$ Un autre sommet de tier 2 pris au hasard
      fin
      Ajouter à G une arête entre S et S_rand avec un poids compris entre 10 et 20
    fin
  fin
fin
Retourner G

la fonction seront donc le sommet et le tier a compter. On aura une fonction de la forme `count_tier(vertex, tier)` où *vertex* correspond au sommet à tester et *tier* la chaîne de caractère du tier à compter. On pourra ainsi faire `count_tier('T2_4', 'T2')` et la fonction retournera le nombre de voisins de tier 2 que comprend le sommet $T2\_4$.

La liste de condition pour trouver un sommet aléatoire $T2$ adéquat à rattacher au sommet S de la boucle est un peu lourd à mettre en place en python. C'est pourquoi il est préférable dans cette situation de créer une autre fonction qui vérifiera que ce sommet respecte bien toute les conditions. Cette fonction a été implémentée dans le projet sous le nom `check_vertex_tier` et prend en paramètre :

- `vertex_name` $\rightarrow$ le nom du sommet à qui on veux donner un voisin de tier 2,
- `rand_vertex` $\rightarrow$ Le nom du sommet pris au hasard parmi les sommets d'un tier x,
- `tierToCheck` $\rightarrow$ Le tier x concerné par la condition, ici T2,
- `neighbor_count` $\rightarrow$ Le nombre de voisin(s) "admis", ici 3 puisqu'on ne peux pas rattacher plus de 3 voisins de tier 2.

Ainsi, quand on fait `check_vertex_tier('T2_3','T2_5', 'T2', 3)`, on demande de vérifier que $T2\_3$ et $T2\_5$ ne sont pas égaux, que $T2\_3$ n'a pas déjà pour voisin $T2\_5$ et que $T2\_5$ a strictement moins de trois voisins de tier 2.

2.3 Tier 3

La création de tier 3 est très similaire à la création de tier 2. Seules les valeurs changent par rapport à la génération de sommet de tier 2:

- création de 70 noeuds,
- connections :
  - à 2 noeuds de tier 2
  - à 1 noeud de tier 3
- ces liaisons seront évalués entre 15 et 50

A noter ici qu'il est possible que deux tiers 3 soient liés à un même tier 2. Il n'y a donc pas besoin de faire toute la vérification faite précédemment pour la connexion tier 2 - tier 2. En revanche pour la connexion de tier 3 à tier 3, il faudra vérifier que le tier 3 pris au hasard n'est pas le même que le tier actuel et qu'il n'est pas déjà connecté à un autre tier 3.

La création de ces noeuds ne nécessite pas plus de fonction que pour la création du tier 2. L'algorithme est très similaire au précédent, les seuls changements se résument au nombre de sommets produits et à la boucle qui créait pour un sommet de tier 2, 2 ou 3 liaisons avec un autre tier 2. Elle n'existe pas ici puisqu'un tier 3 n'a qu'une seule autre liaison avec un autre tier 3. On ajoutera aussi dans la boucle qui parcours les sommets de tier 3 une boucle à 2 itérations qui viendra lier le sommet S actuel et un sommet de tier 2 pris au hasard. L'algorithme ci-dessous correspondant (algorithme 3).

Pour le programme Python, on reprend ici ce qu'on avait fait pour la création des opérateurs de transit et on vient ajouter les modifications vu plus haut.

2.4 Création de tous les tiers

Pour faciliter la génération du graphe avec ces différents tiers, nous avons créer une fonction `CreateTier(G)` qui vient simplement faire appel aux fonctions de création de tier 1, 2 et 3.

3 Vérification de la connexité du réseau

Afin de vérifier que notre graphe est connexe, on va venir faire un parcours en largeur et vérifier que tous les sommets du graphe se trouve dans l'arbre obtenu. L'implémentation d'un *Depth first search* en python est assez simple. Notre fonction `dfs` est récursive. L'idée est de créer une liste de sommets visités et d'ajouter un sommet, parmis les voisins du sommet visité, si celui-ci n'a pas déjà été visité.

Algorithme 3 : Création des opérateurs de tier 3

Entrées : Graphe G contenant les sommets de tier 1 et de tier 2
Résultat : Graphe G avec les sommets de tier 3
pour i *allant de 1 à 70 compris* **faire**
  Ajouter à G le sommet $(T3\_i)$; `// ` i ` = ` $i^{ème}$ ` itération de la boucle`
fin
pour chaque *sommet* S *de tier 3 du graphe* G **faire**
  pour j *allant de 0 à 2* **faire**
    Ajouter une arête entre S et un sommet de tier 2 pris au hasard
  fin
  si S *n'est pas déjà connecté à un autre* $T3$ **alors**
    $S\_rand =$ sommet de tier 3 pris au hasard
    tant que *le sommet* S *est égal au sommet* S_rand
    OU que S_rand *est déjà connecté à 1 autres tier 2*
    OU que S *est déjà lié à* S_rand **faire**
      $S\_rand =$ Un autre sommet de tier 3 pris au hasard
    fin
    Ajouter à G une arête entre S et S_rand avec un poids compris entre 15 et 50
  fin
fin
Retourner G

Ensuite, pour savoir si le graphe est connexe, il suffit de parcourir les sommets du graphe et de vérifier qu'ils sont bien tous dans la liste créer par le parcours en largeur. Si ce n'est pas le cas, alors l'arbre n'est pas connexe, sinon il l'est. l'algorithme du parcours en profondeur et celui de la vérification de la connexité sont ici séparés comme dans le programme python. Ci-dessous les algorithmes de parcours en profondeur (Algorithme 4) et de vérification de connexité (algorithme 5).

Algorithme 4 : Parcours en profondeur

Entrées : Liste $visité$, Sommet S
Résultat : Liste $visité$ contenant tous les sommets visités
si S *n'est pas dans* $visité$ **alors**
  Ajouter à la liste S **pour chaque** *Voisins* V *de* S **faire**
    Parcours en profondeur($visité$,V)
  fin
fin
Retourner $visité$

L'implémentation en Python est assez simple puisqu'on a uniquement besoin des voisins d'un sommet qu'on obtient en entrant `G.vertices['sommet']`.

Algorithme 5 : Vérification de la connexité

Entrées : Graphe G
Résultat : Booléen qui indique VRAI si le graphe est connexe, FAUX sinon

```
/* A contient l'arbre couvrant de G, on donne arbitrairement une
   valeur de départ à la fonction                                  */
```

Liste A = Parcours en profondeur (G, 'T1$'_1$)
pour chaque *sommet* S *de* G **faire**
  si S *n'est pas dans* A **alors**
    Retourner $FAUX$
  fin
fin
Retourner $VRAI$

4 Création de la table de routage - Dijkstra

La table de routage de chaque nœud est calculée grâce à l'algorithme de Dijkstra. On se trouve ici dans une situation qui est adéquate à l'utilisation de cet algorithme puisque tous les sommets sont positifs et qu'on ne cherche pas à recalculer à chaque demande le plus court chemin entre deux sommets. En effet, nous créerons tout d'abord la table de routage, puis nous ferons appel aux prédécesseurs ainsi qu'au tableau de plus court chemin obtenu avec Dijkstra pour trouver le chemin de poids minimum entre deux sommets donnés.

L'algorithme se déroule de la façon suivante: on initialise tous les sommets à ∞ et on met le sommet de départ à 0. On parcours ensuite tous les sommets puis à chaque voisin, on vient regarder si le chemin du sommet de départ au sommet voisin (en passant par le sommet actuel) est inférieur au précédent. Si c'est la cas, alors on vient ajouter cette variable à `d[]` qui stocke "l'accumulation de poids" des sommets. C'est donc un algorithme de Dijkstra classique à la différence qu'on stocke certaines données (les prédécesseurs et le tableau des plus court chemins) pour pouvoir les utiliser plus tard. Plus clairement, voici l'implémentation de l'algorithme de Dijkstra du projet :

Il nous faut maintenant prendre les variables `predecesseurs` et d pour obtenir le chemin de poids minimum entre deux sommets. L'idée est de remonter les sommets par le biais de la variable `predecesseurs` et d'afficher le poids du chemin graçe à d.

5 Programme

5.1 Fichier principal - network.py

Fichier contenant le programme principal du projet avec le main, la création de tier et la fonction permettant de trouver le plus court chemin entre deux sommets.

```
import random
from graph import Graph
from graph import Vertex
```

Algorithme 6 : Dijkstra

Entrées : Graphe G, Sommet $départ$
Résultat : Liste $prédécesseurs$, Liste d des plus court chemins
`/* Initialisation */`
pour chaque *noeud S de G* **faire**
  $d[S] = \infty$
fin
Liste $nonVisité$ = ensemble de tous les noeuds de G
tant que $nonVisité$ *n'est pas vide* **faire**
  S = sommet de poids minimum en partant de $départ$ enlever S de $nonVisité$
  pour chaque *voisin(s) V de S* **faire**
    si $d[V] > d[S]$ *+ poids entre V et S* **alors**
      $d[V] = d[S]$ + poids entre V et S $predecesseurs[V] = S$
    fin
  fin
fin
Retourner predecesseurs, d

Algorithme 7 : Plus court chemin

Entrées : Liste predecesseurs, Liste d, Sommet $départ$, Sommet $arrivé$
Résultat : Affiche le plus court chemin entre le sommet de $départ$ et le sommet d'$arrivés$
Liste chemin **tant que** *sommet S n'est pas le sommet $départ$* **faire**
  Insérer S dans $chemin$ S = predecesseur[S]
fin
Insérer le sommet $départ$ dans $chemin$ Afficher $d[arrivé]$ Afficher $chemin$
Retourner predecesseurs, d

```python
def main():
    G = Graph()
    G = createTier(G)

    print("\n[USAGE] :\nT1_i  0 < i <= 10\n"
          "T2_k  0 < k <= 20\n"
          "T3_n  0 < n <= 70\n")

    vertexA = input("Enter vertex A : ")
    vertexB = input("Enter vertex B : ")

    if vertexA not in G.vertices or vertexB not in G.vertices:
        print("\nINPUT WARNING : Vertex not in graph, correct input:\nT1_1 ... T1_10\nT2_1 ... T2_20\nT3_1 ... T3_70")
        exit()
```

```
    # G.print()

    if G.check_connected():
        predecessor, shortest_distance = G.dijkstra(vertexA)
        path = shortest_path(predecessor, shortest_distance, vertexA, vertexB)

        # print_tables(predecessor, shortest_distance, G, all)

        if shortest_distance[vertexB] != 99999:
            print('\nShortest distance is ' + str(shortest_distance[vertexB]))
            print('And the path is ' + str(path))
    else:
        print("Graph is not connected - aborting")


"""
Create backbone
"""
def createTier1(G):
    for i in range(1, 11):
        buf = Vertex('T1_'+str(i))
        G.add_vertex(buf)

    tmp_vertices = G.vertices.copy()

    for vertex in G.vertices:
        for vertexNeighbor in tmp_vertices:
            if random.randrange(100) < 75 and vertex != vertexNeighbor:
                G.add_edge(vertex, vertexNeighbor, random.randrange(5, 11))
        tmp_vertices.pop(vertex)  # Remove vertex that has already been treated

    return G


"""
Create transit nodes
"""
def createTier2(G):
    verticesT2 = []  # To only take vertices 'T2' in the gaph and not all vertices
    for i in range(1, 21):
        buf = Vertex('T2_'+str(i))
        G.add_vertex(buf)
```

```
        verticesT2.append(buf.name)

    for vertexT2 in verticesT2:  # loop on all 'T2'
        for i in range(0, random.randrange(1, 3)):
            G.add_edge(vertexT2, G.pick_random_vertex(
                'T1'), random.randrange(10, 21))

    for vertex in verticesT2:
        randValue = random.randrange(2, 4)
        for y in range(0, randValue):
            # check if the actual vertex hasn't been treated yet
            if G.count_tier(vertex, 'T2') < 3 and randValue != G.count_tier(vertex, 'T2'):
                rand_vertex = G.pick_random_vertex('T2')

                while G.check_vertex_tier(vertex, rand_vertex, 'T2', 3):
                    rand_vertex = G.pick_random_vertex('T2')

                G.add_edge(vertex, rand_vertex, random.randrange(10, 21))
    return G


"""
Create nodes of tier 3
"""
def createTier3(G):
    verticesT3 = []
    for i in range(1, 71):
        buf = Vertex('T3_'+str(i))
        G.add_vertex(buf)
        verticesT3.append(buf.name)

    for vertex in verticesT3:
        for i in range(2):
            G.add_edge(vertex, G.pick_random_vertex(
                'T2'), random.randrange(15, 50))

        if G.count_tier(vertex, 'T3') < 1:
            rand_vertex = G.pick_random_vertex('T3')
            while G.check_vertex_tier(vertex, rand_vertex, 'T3', 1):
                rand_vertex = G.pick_random_vertex('T3')
            G.add_edge(vertex, rand_vertex, random.randrange(15, 50))

    return G

```

```python

"""
Create all tiers
"""


def createTier(G):
    G = createTier1(G)
    print("Tier 1 created")
    G = createTier2(G)
    print("Tier 2 created")
    G = createTier3(G)
    print("Tier 3 created")
    return G


"""
Find the shortest path, based on what Dijkstra found for predecessors and the
shortest distance
"""
def shortest_path(predecessor, shortest_distance, start, end):
    path = []
    currentVertex = end
    while currentVertex != start:
        try:
            path.insert(0, currentVertex)
            currentVertex = predecessor[currentVertex]
        except KeyError:
            break
    path.insert(0, start)
    return path


"""
print the routing table of aked tier
"""
def print_tables(predecessor, shortest_distance, G, tier):
    for vertex in G.vertices:
        print("\n| // ", vertex, " //")
        for destination in G.vertices:
            if destination != vertex:
                path = shortest_path(
                    predecessor, shortest_distance, vertex, destination)
                print("|", destination)
                print("|", path)
```

```python


if __name__ == "__main__":
    main()
```

5.2 Fichier structure - graph.py

Fichier contenant les fonctions utiles sur les graphes et les sommets.

```python
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import random
import sys
import collections

class Vertex:
    def __init__(self, vertex):
        self.name = vertex
        self.neighbors = {}

    """
    Add neighbor to actual vertex
    Add actual vertex to neighbors of neighbor
    """
    def add_neighbor(self, neighbor, weight):
        if isinstance(neighbor, Vertex): # check if neighbor is a Vertex
            if neighbor.name not in self.neighbors and (neighbor.name != self.name): # if neighbord to add is not already in neighbors list
                self.neighbors[neighbor.name] = weight
                neighbor.neighbors[self.name] = weight
        else :
            return false

class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, vertex):
        if isinstance(vertex, Vertex):
            self.vertices[vertex.name] = vertex.neighbors
        else:
            return false

    def remove_vertex(self, vertexToRemove):
```

```
        self.vertices.pop(vertexToRemove)
        for vertexNeighbor in self.vertices.keys():
            if vertexToRemove in self.vertices.get(vertexNeighbor):
                self.vertices.get(vertexNeighbor).pop(vertexToRemove)

    def remove_edge(self, vertexA, vertexB):
        i = 0
        # if there is one element in vertex
        if len(self.vertices.get(vertexA)) == 1:
            self.vertices.pop(vertexA)
        else:
            self.vertices[vertexA].pop(vertexB)

        if len(self.vertices.get(vertexB)) == 1:
            self.vertices.pop(vertexB)
        else:
            self.vertices[vertexB].pop(vertexA)

    """
    Add adge between two vertex name
    Vertex is created and his neighbor is set properly
    """
    def add_edge(self, vertex1_name, vertex2_name, weight):
        self.vertices[vertex1_name].update({vertex2_name: weight})
        self.vertices[vertex2_name].update({vertex1_name: weight})

    """
    param: tier (a String) is the tier chosen (T1, T2 or T3)
    return: Return the name (in String) of a vertex chosen randomly
    """
    def pick_random_vertex(self, tier):
        list_tier = []

        for vertex in self.vertices:
            if tier in vertex:
                list_tier.append(vertex)

        vertex_name = random.choice(list_tier)
        return vertex_name

    """
    Count the number of edge of vertex in specific tier
    param: <vertex>
    """
    def count_tier(self, vertex, tier):
        count = 0
        for k in self.vertices[vertex]:
```

```
            if tier in  k:
                count+=1

        return count

    """
    Check if the vertex took randomly is not the same has the vertex to asign,
          if the vertex took randomly has not already <neighbor_count> neighbor(s) of <tierToCheck>,
          if the vertex took randomly is not already in the vertex
    """
    def check_vertex_tier(self, vertex_name, rand_vertex, tierToCheck, neighbor_count):
        equal = vertex_name == rand_vertex  # Check if vertex are equal
        neighbor_tier_count = self.count_tier(rand_vertex,tierToCheck) >= neighbor_count   # Count how many tier are already connected to the neighbor
        inVertex = False  # Check if the vertex took randomly is not already the neighbor of the actual vertex

        if rand_vertex in self.vertices[vertex_name]:
            inVertex = True

        if equal or neighbor_tier_count or inVertex:
            return True
        else:
            return False

    """
    Simple recursive Depth first search
    """
    def dfs(self, visited, vertex):
        if vertex not in visited:
            #print(vertex)
            visited.add(vertex)
            for neighbor in self.vertices[vertex]:
                self.dfs(visited, neighbor)
            return visited

    """
    Return true if the graph is connected, false if not
    """
    def check_connected(self):
        spanning_tree = set()
        spanning_tree = self.dfs(spanning_tree, 'T1_1')

```

```
        for vertex in self.vertices:
            if vertex not in spanning_tree:
                return False

        return True

    """
    Simple implementation of Dijkstra algorithm
    """
    def dijkstra(self, start):
        shortest_distance = {}
        predecessor = {}
        unseenVertices = self.vertices.copy()
        infinity = 99999

        for vertex in unseenVertices:
            shortest_distance[vertex] = infinity
        shortest_distance[start] = 0

        while unseenVertices:
            minVertex = None
            for vertex in unseenVertices:
                if minVertex is None:
                    minVertex = vertex
                elif shortest_distance[vertex] < shortest_distance[minVertex]:
                    minVertex = vertex
            for childVertex, weight in self.vertices[minVertex].items():
                if weight + shortest_distance[minVertex] < shortest_distance[childVertex]:
                    shortest_distance[childVertex] = weight + shortest_distance[minVertex]
                    predecessor[childVertex] = minVertex
            unseenVertices.pop(minVertex)
        # print(shortest_distance)

        return predecessor, shortest_distance


    """
    Just a simple way to display the graph graphically - not essential
    useless on big graph
    """
    def display(self):
        FROM = []
        TO = []
```

```

        for vertex in self.vertices:

            for vertexNeighbor in self.vertices[vertex]:
                FROM.append(vertex)
                TO.append(vertexNeighbor)

        df = pd.DataFrame({ 'from': FROM, 'to': TO})
        G=nx.from_pandas_edgelist(df, 'from', 'to')
        nx.draw(G, with_labels=True, node_size=1000, node_color="
    skyblue", pos=nx.fruchterman_reingold_layout(G))
        plt.show()


    """
    print in terminal the current graph, node by node
    with the number of connexion current node has
    """
    def print(self):
        for vertex in self.vertices:
            print("")
            print(vertex + " T1:" + str(self.count_tier(vertex,'T1')) +
     " /T2: " + str(self.count_tier(vertex,'T2')) + " /T3:" + str(self.
    count_tier(vertex,'T3')))
            print(self.vertices.get(vertex))
```