Rapport de Projet – Tables de routage (2023/2024)

IN403 Algorithmique

Projet réalisé en binôme :

- **MACHTER Massinissa 22304222**
- **KHAZEM Lynda 22304263**
- **Groupe TD04**

Préambules :

- Il faut s'assurer d'avoir les packages python **(Tkinter , matplotlib et networkx)** sur votre version python ,sinon il faut les installer avec la commande « pip –m install 'package' » .

- Pour lancer notre application, vous avez qu'à exécuter le module **« application.py »**

1. Introduction :

Le projet vise à concevoir une application permettant de générer et de gérer les tables de routage de chaque nœud d'un réseau de 100 nœuds. À travers ce rapport, nous décrirons les différentes étapes de développement de cette application, en mettant l'accent sur les aspects techniques (partie de code) et les défis rencontrés**.**

2. La Création Aléatoire d'un Réseau Réaliste :

La première tâche consiste à créer la topologie d'interconnexion d'un réseau de 100 nœuds. Cette topologie est composée d'un backbone de 10 nœuds très connectés entre eux, 20 opérateurs de niveau 2 (Tier 2) et le reste d'opérateurs de niveau 3 (Tier 3). Les liens et les temps de communication sur chaque lien sont générés de manière aléatoire, en respectant des règles spécifiques (voir le sujet).

Pour déterminer **la structure de données** la plus appropriée pour représenter et mémoriser le graphe correspondant à notre réseau de communication, nous avons opté pour deux classes principales : **Nœud()** et **Graphe().**

2.1 La classe Noeud :

Elle a été conçue pour représenter chaque nœud du réseau. Chaque instance de cette classe contient des attributs comme l'identifiant du nœud, une indication s'il s'agit d'un nœud de niveau 2, la liste de ses voisins et sa table de routage. Les méthodes de cette classe permettent d'ajouter des voisins, de calculer la distance entre deux nœuds, de récupérer la liste des voisins, de vérifier si un nœud est voisin, et de compter le nombre de voisins de niveau 2.

```
class Noeud:
    ''' cette classe modelise les objets qui sont des noeuds (sommets)'''

    def __init__(self, identifiant,tier2=False):
        ''' notre graphe il est Non oriente donc on parle de liste des voisins d'un noeud '''
        self.id = identifiant
        self.estTier2=tier2
        self.voisins = []
        self.table_routage = {}

    def ajouter_voisin(self, voisin, temps_communication):
        ''' permet l'ajout d'un noeud voisin ainsi que le poid de l'arete
        dans la liste des voisins '''

        self.voisins.append((voisin, temps_communication))

    def distance(self,noeud):
        ''' retourne la distance entre ce sommet(self) et son voisin(noeud)'''

        for voisin,temps_communication in self.voisins:
            if voisin == noeud:
                return temps_communication

    def mes_voisins(self):
        ''' return la liste des voisins '''
        if self.voisins :
            voisins,_=zip(*self.voisins)#unzip list(v,w)
        else:
            voisins=[]
        return list(voisins)

    def est_mon_voisin(self,noeud):
        ''' verifié si un noeud donné est un voisin '''

        for voisin,t in self.voisins:
            if noeud == voisin:
                return True
        return False

    def nb_voisins_tier2(self):
        ''' retourne le nombre de voisins qui sont tier2 '''

        return len([ v for v in self.mes_voisins() if v.estTier2==True])
```

Nb : nous avons rajouté l'attribut estTier2 qui est a True que pour un nœud Tier2 et la méthode nb_voisins_tier2 (), car les operateurs de niveau 2 sont les seuls qui sont reliés aux 2 autres niveaux d'operateurs conformément aux règles spécifiées

2.2 La classe Graphe

Elle a été élaborée pour représenter le graphe complet du réseau. Elle contient la liste de nœuds et des méthodes pour ajouter un nœud au graphe, générer le réseau selon les règles spécifiées, générer les tables de routage de chaque nœud, et reconstruire le chemin entre deux nœuds afin de transmettre le message en suivant le plus court chemin en temps de communication .

```
class Graphe:
    ''' cette classe modelise des objets graphe '''

    def __init__(self):
        ''' le graphe est represente par la liste de ses noeuds ( instances de classe Noeud)'''
        self.noeuds = []

    def ajouter_noeud(self, noeud):
        ''' ajout d'un noeud dans la liste des noeuds '''
        self.noeuds.append(noeud)
```

la méthode **generer_reseau()** de la classe Graphe est chargée de créer le réseau , en respectant les règles spécifiées dans le sujet du projet :

- Les 10 premiers nœuds sont considérés comme faisant partie du backbone, Chaque paire de nœuds dans le backbone a une chance de 75% (random()<0.75) d'être connectée, avec des liens évalue entre 5 et 10 unités.

```
    def generer_reseau(self):
        ''' Generation de resaux d'interconnexion d'un graphe de 100 noeuds,
        - Les liens et les temps de communication sur chacun de ces liens vont etre
        crees de maniere aleatoire selon des regles donné (voir sujet)'''

        # Génération du backbone (Tier 1)

        backbone = [Noeud(i) for i in range(1,11)]
        for i in range(len(backbone)):
            for j in range(i+1, len(backbone)):
                if random.random() < 0.75:  # 75% de chance de création du lien
                    temps_communication = random.randint(5, 10)
                    backbone[i].ajouter_voisin(backbone[j], temps_communication)
                    backbone[j].ajouter_voisin(backbone[i], temps_communication)
```

- 20 nœuds sont créés pour représenter les opérateurs de niveau 2.Chaque opérateur de niveau 2 est connecté à 1 ou 2 nœuds du backbone de manière aléatoire. Ensuite, chaque opérateur de niveau 2 est connecté à 2 ou 3 autres opérateurs de niveau 2 de manière aléatoire
- Dans cette partie nous devons respecter plusieurs contraintes a savoir deux operateur Tier2 ne sois pas relier entre eux plusieurs fois, c'est pour cela on a implémenter la méthode nb_voisins_tier2 () dans la classe Nœud, pour qu'a chaque fois qu'on doit le relier avec un autre operateur Tier2 tiré aléatoirement

on vérifie a la fois si il n'est pas déjà son voisin (Méthode est_mon_voisin()) et si leurs nombre de voisins Tier2 est au maximum 3.

```
# Génération des opérateurs de niveau 2 (Tier 2) 20 operateurs

tier2 = [Noeud(i,True) for i in range(11, 31)]
for noeud2 in tier2:
    # Connecter à 1 ou 2 nœuds du backbone
    # tiré aleatoirement
    x_backbone=random.randint(1,2)
    while x_backbone>0:
        j=random.randint(0,9)
        if not(noeud2.est_mon_voisin(backbone[j])):
            temps_communication = random.randint(10, 20)
            noeud2.ajouter_voisin(backbone[j], temps_communication)
            backbone[j].ajouter_voisin(noeud2, temps_communication)
            x_backbone-=1

    # on tire 2 ou 3 operateurs niv 2 aleatoirement
    x_tier2=random.randint(2,3)

    while noeud2.nb_voisins_tier2()<x_tier2:
        j=random.randint(0,19)
        # si le noeud n'est pas deja son voisin (graphe simple);
        # la derniere condition pour eviter les boucle (s,s)
        if not noeud2.est_mon_voisin(tier2[j]) and  tier2[j].nb_voisins_tier2()<3 and tier2[j]!=noeud2:
            temps_communication = random.randint(10, 20)
            noeud2.ajouter_voisin(tier2[j], temps_communication)
            tier2[j].ajouter_voisin(noeud2, temps_communication)
```

- Les 70 nœuds restants sont des opérateurs de niveau 3, Chaque opérateur de niveau 3 est connecté à 2 opérateurs de niveau 2 de manière aléatoire, on a juste a vérifier qu'il n'est pas déjà son voisin (graphe simple)

```
# Génération des opérateurs de niveau 3 (Tier 3)
tier3=[Noeud(i) for i in range(31,101)]
for noeud3 in tier3:
    # relié l'operateur tier3 a 2 opé tier2 tiré aleat
    # avec des lien evalué entre 20 et 50
    while len(noeud3.mes_voisins())<2:
        j=random.randint(0,19)
        if not(noeud3.est_mon_voisin(tier2[j])):
            temps_communication = random.randint(20,50) # evaluation de l'arete
            tier2[j].ajouter_voisin(noeud3, temps_communication)
            noeud3.ajouter_voisin(tier2[j], temps_communication)
```

A la fin de la méthode generer_reseau() , Tous les nœuds créés sont ajoutés à la liste des nœuds du graphe.

```
    # Ajout des nœuds(tous les operateurs) au graphe
    self.noeuds=backbone+tier2+tier3
```

3. La Vérification de la Connexité du Réseau

Même si le réseau est généré de manière aléatoire il est très probable qu'il soit connexe, il est crucial de s'assurer. Pour cela, nous avons développé une procédure de vérification de la connexité du réseau dans le module 'connexite.py'. Cette procédure garantit que le graphe

est connexe lorsque tous les sommets peuvent être atteints en effectuant un parcours en profondeur à partir de n'importe quel sommet de départ tiré aléatoirement.

Connexite.py :

```python
import random

def pp(s,visi):
    ''' procedure parcours en profondeur (pp) ,
    prend comme argument :
    sommet de depart ,dictionnaire {sommet:booleén(visite ou non)}
    -> fait le pp a partir de s
    '''
    visi[s]=True
    for (v,_) in s.voisins:
        if not visi[v]:
            pp(v,visi)

def connexe(graphe):
    ''' argument:graphe en question --> True si Connexe ,False sinon
    ->si tous les sommets du réseau ont été visités cela signifie que le
    réseau est connexe.Sinon, cela signifie qu'il existe au moins un sommet qui n'est pas accessible à
    partir du sommet de départ choisi, cad le réseau est non connexe.
'''
    # initialisation de tt les sommets a Non Vu(non visité :)
    visites={noeud:False for noeud in graphe.noeuds}
    # le sommet de depart tiré aleat
    x=random.randint(0,len(graphe.noeuds)-1)
    sommet_depart=graphe.noeuds[x]
    # lancer le pp
    pp(sommet_depart,visites)
    return all(visites.values())# voir si tout les sommet ont ete visité ou non
```

Nb : Ici connexe qui lance le premier appel a la fonction pp(s) avec le sommet de départ s, a défaut le graphe n'est pas connexe on lance la génération d'un autre réseau.

4. La Détermination de l'arborescence de PCC en chaque nœud avec Djikstra :

La procédure Djikstra (Djikstra.py) que nous avons implémentée est cruciale pour le projet car elle permet de calculer le plus court chemin depuis un nœud source vers tous les autres nœuds du graphe

Init Djikstra :

```python
def Djikstra(source,graphe):
    ''' calcule le pcc depuis la source vers tous les autres
        sommets de graphe.
        -> fait de ( One to All )
        -> retourne: l'arborescence de plus court chemin depuis source '''
    # Init l'ensemble des sommets deja traité
    T={source}
    # D la liste contenat des tuples ( sommet "v" , distMin(source,v),pere(v) )
    # concraitement , D est la liste contenant l'arborce de pcc a partir de la source
    D=[(source,0.0,None)]
    infini = float("inf")

    for noeud in graphe.noeuds :
        if source.est_mon_voisin(noeud):
            # si l'arete (source,noeud) existe
            D.append((noeud,source.distance(noeud),source))
        else:
            # si c'est pas son voisin ,alors "distance a Infinie" et "pere a None"
            D.append((noeud,infini,None))

    # l'ensemble des sommets de graphe
    V={v for v in graphe.noeuds }
```

Boucle principale : À chaque itération, on sélectionne le nœud non encore traité avec la plus petite distance dans D et on l'ajoute à la liste des nœuds déjà traité T, Ensuite, pour chaque voisin non encore traité de ce nœud sélectionné, l'algorithme met à jour la distance minimale dans D si un chemin plus court est trouvé en passant par le nœud sélectionné.

Suite et fin Djikstra :

```
while T!=V:
    # choisir le noeud avec comme distance di minimale qui...
    #...n'est pas encore traité

    Dtemp=[(s,di,p) for (s,di,p) in D if  s not in T]
    Dtemp.sort(key=lambda x :x[1])
    noeud=Dtemp[0]
    # l'ajouter a la liste des noeud deja traité
    T.add(noeud[0])

    for v in noeud[0].mes_voisins():
        if v not in T :
            for (s,d,p) in D :
                if s==v:
                    break
            nouvelleDist=noeud[0].distance(v)+noeud[1]
            # la distance est améliorer si elle est plus petite que d
            if  nouvelleDist<d:
                D[D.index((v,d,p))] = (v, nouvelleDist, noeud[0])

return D
```

5. La Détermination de la Table de Routage de Chaque Nœud :

Une fois la connexité du réseau est confirmée, nous procédons à la détermination des tables de routage de chaque nœud. Chaque table de routage indique le prochain nœud sur le chemin le plus court vers une destination donnée. Cela implique le développement d'un algorithme efficace pour calculer ces 100 tables de routage :

Il s'agit de la méthode **generer_tables_routage()** dans la classe Graphe(),qui pour chaque nœud de graphe lance une et une seule fois le calcule de pcc avec Djikstra, et établie sa table de routage :

```
def generer_tables_routage(self):
    ''' Methode qui lance en chaque noeud le calcule des tables de routages des noeuds de graphe.
        --> calcule la table de routage d'un noeud pour chaque destination possible (les 99 autres noeuds),
        en utilisant biensur Djikstra pour avoir le plus court chemin en temps de communication'''

    for source in self.noeuds:
        # recuperer l'arborescence de pcc depuis noeud source
        resultats_dijkstra = Djikstra.Djikstra(source, self)

        table_routage = {}

        for info in resultats_dijkstra:
            # info un tuple contient (noeud, distance, pere)
            if info[0] != source:
                destination = info[0] # distination
                distanceMin=info[1] # distance min en unites
                pere = info[2]
                chemin=[]
                while pere != source:
                    chemin.append(pere)
                    pere = [x[2] for x in resultats_dijkstra if x[0] == pere][0]

                if chemin!=[]:
                    table_routage[destination] = (chemin[-1],distanceMin)# Le prochain nœud à atteindre
                else:
                    # donc le pc chemin c'est l'arete directe entre eux
                    table_routage[destination] = ("fin",distanceMin)
        source.table_routage = table_routage
```

6. La Reconstitution du Chemin entre 2 Nœuds :

Enfin, pour permettre à l'utilisateur de saisir deux nœuds, un nœud émetteur et un nœud destinataire du message.Nous utilisons les tables de routage déjà établie pour reconstruire le chemin le plus court entre ces deux nœuds. Cela permet à l'utilisateur de visualiser le chemin emprunté par un message à travers le réseau. Ici il s'agit de la méthode

reconstruire_chemin() de la classe Graphe() :

```
def reconstruire_chemin(self,source, destination):
    ''' methode de reconstruction de chemin entre une source et une destionation de message.
        Rq: la reconstitution de chemin se fait sans refaire le calcul de plus court chemin
        mais juste en exploitant les tables de routage deja etablies'''

    distanceMin=source.table_routage.get(destination)[1]
    chemin = [source.id]
    noeud_actuel = source

    while noeud_actuel != destination:
        if noeud_actuel.table_routage.get(destination)[0] == "fin":
            break
        prochain_noeud = noeud_actuel.table_routage[destination][0]
        chemin.append(prochain_noeud.id)
        noeud_actuel = prochain_noeud

    chemin.append(destination.id)
    return chemin,distanceMin
```

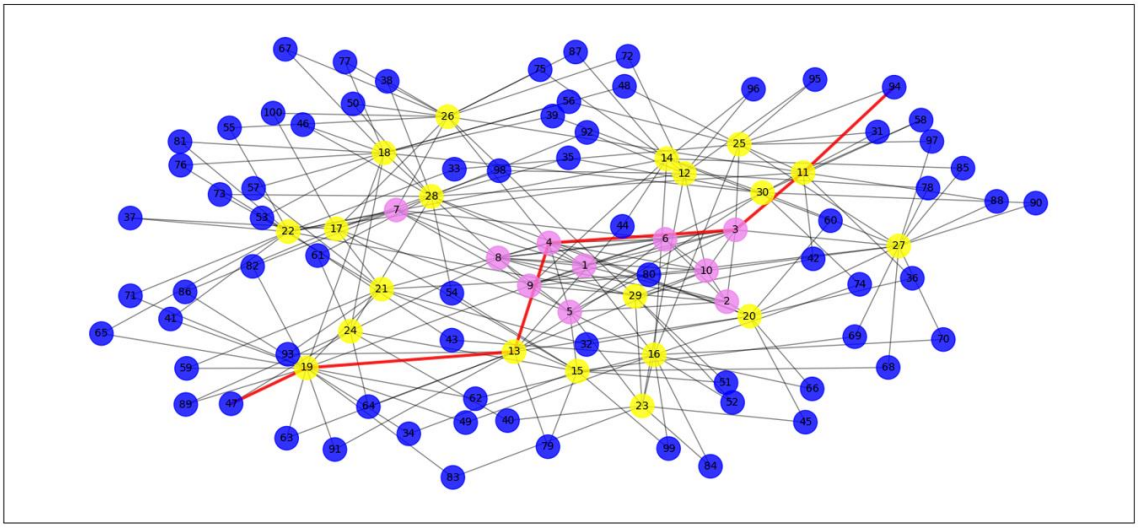
Ici la méthode retourne le tuple qui contient le pcc qui est une liste de avec les id des Nœud à suivre jusqu'à la destination et la distance Minimale vers la destination.

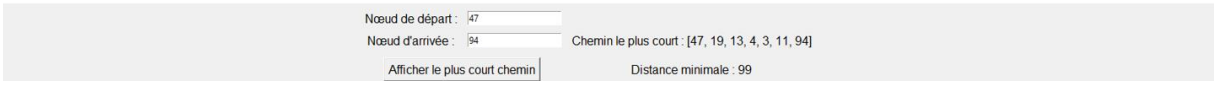
7. Interface graphique :

L'interface graphique est conçu pour visualiser le graphe, tier1 sont en violet, tier2 sont en jaune, et les tier3 sont en bleu. L'utilisateur est invité à choisir un nœud de départ et un nœud destination puis clique sur afficher le plus court chemin. Le pcc s'affiche en rouge sur le graphe, en mentionnant la liste de pcc ainsi que la distance Minimale. L'utilisateur peut réaliser plusieurs recherches de plus court chemin sur le même graphe depuis et vers n'importe quel nœud du graphe (numérotée de 1 à 100).

Cette interface est faite avec en utilisant les packages Tkinter , matplotlib et networkx de python .

Exemple d'utilisation :





8. Conclusion

Ce projet nous a permis de plonger profondément dans le domaine des réseaux d'interconnexion et de mettre en pratique des concepts de la Théorie des graphes et réseaux informatique tels que les tables de routage et les algorithmes de recherche de plus court chemins. En développant cette application, nous avons acquis une compréhension approfondie des défis associés à la gestion des réseaux informatiques.