ALGORITHMIQUE DE GRAPHES

RAPPORT DE PROJET

HAMMADI Bahaa Eddine

BENARBIA Mohamed Idris

2.1 Description du cadre du projet :

Ce projet vise à explorer les fondements de l'infrastructure Internet en se concentrant sur quelques concepts clés. Le Transit IP est la voie royale pour transporter les données d'un endroit à un autre sur Internet, comme un réseau routier mondial. Le Backbone, quant à lui, est l'autoroute principale de cette infrastructure, reliant les réseaux régionaux et locaux pour former le squelette de l'Internet.
Le rôle des Opérateurs de Niveau (Tier) est essentiel dans cette histoire, classés en différentes couches en fonction de leur taille et de leur connectivité. Du niveau 1, les géants sont directement connectés au Backbone, tandis que les niveaux inférieurs dépendent des niveaux supérieurs pour le transit, chacun contribuant à la toile numérique.

Enfin, le Peering intègre une dimension d'entraide et de collaboration dans cette saga technologique. Les opérateurs ont convenu d'échanger du trafic directement entre leurs réseaux, ce qui réduit la dépendance aux tiers et améliore la performance globale du réseau.

En somme, ce projet nous plonge dans les coulisses de l'Internet, révélant les rouages complexes qui permettent à notre monde interconnecté de fonctionner de manière fluide et efficace.

Description du programme:

Création de la Topologie de Réseau :
Nous avons mis en place la topologie du réseau en utilisant des nœuds représentant différents niveaux d'opérateurs (Tier). Cette topologie comprend un backbone (Tier 1) fortement connecté, des opérateurs de niveau 2 (Tier 2) connectés au backbone et entre eux, et des opérateurs de niveau 3 (Tier 3) connectés aux opérateurs de niveau 2.

Vérification de la Connexité du Réseau :
Nous avons implémenté une procédure pour vérifier que tous les nœuds du réseau sont accessibles à partir d'un nœud de départ quelconque. En cas de non-connexion, nous avons ajouté une fonction pour recréer un nouveau réseau connecté.

Détermination des Tables de Routage :

Nous avons utilisé l'algorithme de Dijkstra pour calculer les chemins les plus courts entre chaque nœud et tous les autres nœuds du réseau. Ces chemins ont été enregistrés dans des tables de routage pour chaque nœud.

Reconstitution du Chemin entre Deux Nœuds :

Nous avons développé une fonctionnalité permettant à l'utilisateur de saisir deux nœuds et de reconstituer le chemin que suivrait un message entre ces deux nœuds en utilisant les tables de routage établies.

Structure de données et Procédures :

Noeud :

Il s'agit d'une classe pour représenter les noeuds du réseau, chaque noeud a un identifiant et un dictionnaire nommé “connection” qui contient tous ses successeurs comme clefs et qui ont pour valeur le temps de communication entre ces 2 noeuds.

Méthode de la classe Noeud :

- **__init__(self, id_noeud)** : contructeur de la classe et initialisation du dictionnaire vide
- **ajouter_connection (self, noeud, temps_communication) :** ajoute un arc entre l'instance et un autre noeud ainsi que le temps de communication entre eux .
- **set_connection(self, connection) :** mutateur de l'attribut connection (le dictionnaire)
- **get_connection(self) :** getter de l'attribut connection .
- **set_id_noeud(self, id_noeud) :** mutateur de l'attribut id_noeud .
- **get_id_noeud(self) :** getter de l'attribut id_noeud .

Et nous avons fini notre classe par encapsuler nos variables d'instance

Programme principale:

- **Créer réseau () :**

Une procédure qui ne prend rien en paramètres, et comme son nom l'indique elle crée notre fameux réseau constitué de 100 Noeuds. On commence par créer 3 listes (une liste pour chaque tier) contenant des instances de la classe noeud (10, 20 et 70 respectivement par niveau croissant), Ensuite on établit les connections entre les noeuds comme demandé dans l'énoncé.

Les noeuds du BACKBONE sont à 75% connecté entre eux.
Un noeud du tier2 est connecté avec 1 ou 2 noeuds du BACKBONE et avec 2 ou 3 noeuds de son opérateur.
Les noeuds du tiers 3 sont connecté avec 2 noeuds exactement de niveau 2.

À la fin, la procédure renvoie la concaténation des ces 3 listes.

est_connecté (réseau) :

Une procédure qui prend en paramètre le réseau et vérifie sa connexité.
Elle prend un noeud de départ (pour nous c'est le noeud 1), et un set (ensemble) noté visité.
En python, la particularité des ensembles est qu'ils n'acceptent pas les redondances (l'ajout d'un élément qui existe déjà ne fait rien).
À l'intérieur de cette procédure on trouve une fonction parcours_profondeur (noeud) qui prend un noeud comme argument et effectue un parcours en profondeur, On lui passe notre noeud de départ et chaque noeud rencontré on le rajoute dans l'ensemble.

À la fin on compare la taille du réseau avec la taille du set.

Dijsktra (graphe, noeud) :

L'algorithme de Dijsktra pour calculer le plus court chemin entre le noeud passé en paramètre et les autres noeuds.
Elle a comme viarbales locale :

- **dictionnaire_noeud_distance :** un dictionnaire qui a comme clé le noeud destinataire et pour valeur la distance pour l'atteindre (dans notre cas c'est le temps de communication).
- **dictionnaire_fils_père:** un dictionnaire, pour chaque noeud destinataire on mémorise son premier antécédent.
- **liste_priorité :** une liste qui va contenir des tuples de type (distance, noeud) qui nous facilite le choix du noeud prochain à traiter (celui avec la distance minimale)

Elle se serve de deux autre fonction aussi :

distance_minimal(liste_distance_noeud) : une fonction qui prend en paramètre la liste de priorité de Dijkstra et renvoie le tuple avec la distance minimale.

ajouter_liste_priorité(distance_calculé, noeud, liste_priorité) : une fonction qui prend en paramètre la nouvelle distance calculée d'un noeud, le noeud en lui même et la liste de priorité.

Si le noeud existe déjà, on modifie sa distance, sinon on le rajoute. (La distance référence toujours le temps de communication).

On Commence par l'initialisation :

Le noeud de départ à pour distance 0 et n'a pas de père. Ses voisins ont pour distance le temps de communication entre eux et le noeud de départ et ce dernier comme père. Pour le reste ils n'ont pas encore de père (None) et ils ont une distance "infinie". On ajoute tuple (0, noeud de départ) à liste_priorité.

Bouclage :
tant que liste_priotié n'est pas vide, On récupère le noeud à traiter, on l'enlève de liste_priorité et pour chaque voisin :

on calcule la nouvelle distance pour l'atteindre et si elle est inférieure à sa distance sauvgardée dans distionnaire_distance_noeud, on lui affecte la nouvelle distance, le nouveau père qui est le noeud traité actuellement et on le rajoute dans liste priorité.

À la fin on retourne l'arborescence obtenue qui distionnaire_fils_père .

calcule_table_routage (réseau) :

Une procédure qui calcule la table de routage de chaque noeud. Elle prend
en paramètre notre réseau, Pour chaque noeud du réseau :

On crée une table de routage vide (c'est un dictionnaire).
Pour fils (différent du noeud actuel), père dans l'arborescence obtenue par dijkstra (réseau, noeud) :

On extrait le premier noeud du chemin pour atteindre ce fils.

saisie_noeud(message) :

Une fonction de contrôle de saisie de l'utilisateur. Elle prend en paramètre un message à afficher et elle force l'utilisateur à saisir un nombre compris entre 1 et 100.

À la fin de notre code source on retrouve le MAIN :

- C'est là où on effectue tous les affichage du terminal
- On force le réseau à être connexe en rebouclant est_connecté (réseau) jusqu'à ce qu'il le devienne.
- On stocke les tables de routage dans fichier texte généré appelé "tables_routage.txt" et on traite la dernière question du projet :
- On récupère les deux noeud de l'utilisateur
- On les repère dans la liste
- On affiche tous les nœuds par lesquels notre message passe.

FIN DE PROGRAMME

```python
import random


class Noeud:
    """
    Class pour représenter un nœud
    """

    def __init__(self, id_noeud):
        """
        Constructeur pour créer un nœud
        """
        self.__id_noeud = id_noeud
        self.__connection = {}

    def ajouter_connection(self, noeud, temps_communication):
        """
        Méthode pour créer une connection entre de nœud
        """
        self.__connection[noeud] = temps_communication

    def set_connection(self, connection):
        """Mutateur de l'attribut connection"""
        self.__connection = connection

    def get_connection(self):
        """Accesseur de l'attribut connection """
        return self.__connection

    def set_id_noeud(self, id__noeud):
        """Mutateur de l'attribut id_noeud"""
        self.__id_noeud = id__noeud

    def get_id_noeud(self):
        """Accesseur de l'attribut id_noeud"""
        return self.__id_noeud

    # Propriété de la classe
    connection = property(get_connection, set_connection)
    id_noeud = property(get_id_noeud, set_id_noeud)


# Question 2.2 : Création de notre joli réseau
def créer_réseau():
    """
    Fonction pour créer le réseau
    """
    # Création des opérateurs pour les 3 niveaux
    tier_01 = [Noeud(i) for i in range(1, 11)]
    tier_02 = [Noeud(i) for i in range(11, 31)]
    tier_03 = [Noeud(i) for i in range(31, 101)]
    # Création du backbone
    for noeud_01 in tier_01:
        for noeud_02 in tier_01:
            if noeud_01 != noeud_02 and random.random() < 0.75:
                temps_communication = random.randint(5, 10)
                noeud_01.ajouter_connection(noeud_02, temps_communication)
                noeud_02.ajouter_connection(noeud_01, temps_communication)

    # Connection des opérateurs de niveau 2 avec niveau 1
    for noeud_02 in tier_02:
        connection_tier01 = random.choices(tier_01, k=random.randint(1, 2))
        temps_communication = [random.randint(10, 20) for i in range(len(connection_tier01))]
        for indice in range(len(connection_tier01)):
            noeud_02.ajouter_connection(connection_tier01[indice], temps_communication[indice])
            connection_tier01[indice].ajouter_connection(noeud_02, temps_communication[indice])

    # Connection des opérateurs de niveau 2 avec niveau 2
    for noeud in tier_02:
        for indice in range(random.randint(2, 3)):
            noeud_02 = random.choice(tier_02)
            while noeud_02 is noeud:
                noeud_02 = random.choice(tier_02)
            temps_communication = random.randint(10, 20)
            noeud.ajouter_connection(noeud_02, temps_communication)
            noeud_02.ajouter_connection(noeud, temps_communication)
```

```
    # Connection des opérateurs de niveau 2 avec niveau 3
    for noeud in tier_03:
        connection_tier02 = random.choices(tier_02, k=2)
        temps_communication = [random.randint(20, 50) for i in range(2)]
        for i in range(2):
            noeud.ajouter_connection(connection_tier02[i], temps_communication[i])
            connection_tier02[i].ajouter_connection(noeud, temps_communication[i])

    return tier_01 + tier_02 + tier_03


def est_connecté(réseau):
    # Choix d'un nœud de départ quelconque
    noeud_départ = réseau[0]

    # Liste pour suivre les nœuds visités
    visité = set()

    # Fonction récursive pour parcourir le graphe en profondeur
    def parcours_profondeur(noeud):
        visité.add(noeud)
        for voisin in noeud.connection:
            if voisin not in visité:
                parcours_profondeur(voisin)

    # Appliquer la recherche en profondeur pour marquer tous les nœuds accessibles
    parcours_profondeur(noeud_départ)

    # Si tous les nœuds ont été visités, le réseau est connexe
    return len(visité) == len(réseau)


def distance_minimale(liste_distance_noeud):
    """
    Fonction qui retourne le nœud avec la distance minimale pour l'atteindre
    """
    distance_min, noeud_min = liste_distance_noeud[0]
    indice = 1
    while indice < len(liste_distance_noeud):
        distance, noeud = liste_distance_noeud[indice]
        if distance < distance_min:
            distance_min = distance
            noeud_min = noeud
        indice += 1
    return distance_min, noeud_min


def ajouter_liste_priorité(distance_calculé, noeud, liste_priorité):
    """
    Fonction qui ajoute un nœud à la liste de priorité avec la distance exigé pour l'atteindre
    """
    indice = 0
    while indice < len(liste_priorité):
        distance, sommet = liste_priorité[indice]
        # Tester si le nœud existe déjà dans la liste
        if sommet == noeud:
            liste_priorité[indice] = distance_calculé, noeud
            return liste_priorité
        indice += 1
    # S'il n'existe pas, on l'ajoute directement
    return liste_priorité + [(distance_calculé, noeud)]


def dijkstra(graph, noeud_initial):
    """
    Fonction qui applique notre fameux algorithme de dijkstra vu en cours
    """
    # Dictionnaire qui stock pour chaque nœud, Sa distance entre lui et le nœud initial
    dictionnaire_noeud_distance = {}
    # Dictionnaire qui stock pour chaque nœud, Son prédécesseur (père)
    dictionnaire_fils_père = {}
    liste_priorité = [(0, noeud_initial)]
    # Initialisation de l'algorithme
    for noeud in graph:
        # les sommets atteignable
        if noeud in noeud_initial.get_connection():
            dictionnaire_noeud_distance[noeud] = noeud_initial.get_connection()[noeud]
            dictionnaire_fils_père[noeud] = noeud_initial
```

```
            liste_priorité.append((dictionnaire_noeud_distance[noeud], noeud))
        else:
            dictionnaire_noeud_distance[noeud] = "infini"
            dictionnaire_fils_père[noeud] = None

    dictionnaire_noeud_distance[noeud_initial] = 0

    while liste_priorité:
        # On récupère le nœud avec la distance minimale
        distance_courante, noeud_actuel = distance_minimale(liste_priorité)
        # Mettre à jour la liste de priorité en fonction des nœuds déjà traités
        liste_priorité.remove((distance_courante, noeud_actuel))
        # Traitement des sommets adjacents du nœud actuel
        for voisin, temps_communication in noeud_actuel.get_connection().items():
            distance = distance_courante + temps_communication
            # Si le nœud n'est pas encore traité ou sa distance peut être réduite , on effectue les modifications
            # nécessaires
            if dictionnaire_noeud_distance[voisin] == "infini" or distance < dictionnaire_noeud_distance[voisin]:
                dictionnaire_noeud_distance[voisin] = distance
                dictionnaire_fils_père[voisin] = noeud_actuel
                liste_priorité = ajouter_liste_priorité(distance, voisin, liste_priorité)
    # On retourne l'arborescence obtenue
    return dictionnaire_fils_père


def calcule_table_routage(réseau):
    tables_routage = {}

    # Pour chaque nœud dans le réseau
    for noeud in réseau:
        # Initialiser la table de routage pour ce nœud
        table_routage = {}
        # Calculer les distances les plus courtes à partir de ce nœud
        arborescence = dijkstra(réseau, noeud)
        # Extraction du premier nœud du plus court chemin
        for fils, père in arborescence.items():
            destination = fils
            if fils != noeud:
                while père != noeud:
                    fils = père
                    père = arborescence[père]
                table_routage[destination] = fils

        tables_routage[noeud] = table_routage
    # Retourner la table de routage pour chaque nœud
    return tables_routage


def saisie_noeud(message: str):
    """
    Fonction qui permet à l'utilisateur d'effectuer une saisie correct
    """
    saisie_incorrect = True
    while saisie_incorrect:
        saisie = input(message)
        try:
            saisie = int(saisie)
            assert 1 <= saisie < 101
            saisie_incorrect = False
        except ValueError:
            print("Veuillez saisir un entier")
        except AssertionError:
            print("Veuillez saisir un numéro compris entre 0 et 100")
    return saisie


if __name__ == "__main__":

    # Création du réseau
    print("Question 2.2 - La création aléatoire d'un réseau réaliste : ")
    réseau = créer_réseau()
    print("Le réseau est bien créé.\n")

    # Vérification de la connexité du réseau
    print("Question 2.3 - La vérification de la connexité du réseau: ")
    if est_connecté(réseau):
        print("Le réseau est connecté.\n")
    else:
        print("Le réseau n'est pas connecté. Création d'un nouveau réseau...")
```

```
    réseau = créer_réseau()
    while not est_connecté(réseau):
        réseau = créer_réseau()
    print("Nouveau réseau créé et connecté.\n")

# Création de la table de routage de chaque nœud du réseau dans un fichier
print("Question 2.4 - La détermination de la table de routage de chaque noeud : ")
tables_routage = calcule_table_routage(réseau)
# Affichage des tables de routage pour chaque nœud
with open("tables_routage.txt", 'w') as fichier:
    for id_noeud, table_routage in tables_routage.items():
        fichier.write(f"Table de routage pour le noeud, {id_noeud.get_id_noeud()} \n")
        for destination, next_hop in table_routage.items():
            fichier.write(
                f" Chemin vers:, {destination.get_id_noeud()}, Prochain sommet:, {next_hop.get_id_noeud()}\n")

print('La table de routage est bien créé dans le fichier: tables_routage.txt\n')

# Reconstitution du chemin entre deux nœuds
print("Question 2.5 - La reconstitution du chemin entre 2 noeuds : ")
noeud1 = réseau[saisie_noeud("Veuillez saisir le premier nœud : ") - 1]
noeud2 = réseau[saisie_noeud("Veuillez saisir le deuxième nœud : ") - 1]
print("Ordre de passage du message:")
while noeud1 != noeud2:
    print(f"Le message est transmis par le noeud : {noeud1.get_id_noeud()}")
    table_routage = tables_routage[noeud1]
    noeud1 = table_routage[noeud2]
print(f"Le message est bien arrivé au noeud {noeud2.get_id_noeud()}")
```