Rapport: Problème du voyageur de commerce

Loïc BERTRAND S4-A2

Projet : Problème du voyageur de commerce

Le projet est intégralement écrit en python 3.8 et il utilise les bibliothèques suivantes :

- prettytable
- termcolor
- numpy
- matplotlib
- progress
- re
- random
- networkx
- time
- os

À priori, sous Linux, il suffit d'exécuter le makefile comme expliqué ci-dessous afin d'installer les bibliothèques python et lancer le programme.

Il faut néanmoins que le gestionnaire de paquet pip pour python 3 soit installé sur votre machine.

Installer le projet

```
make install
```

Lancer le programme

```
make run
```

Utiliser le programme

La navigation se fait via la saisie de chiffres et à la touche Entrer

Graphe sous format fichier

Vous pouvez ajouter ou supprimer des graphes dans le répertoire "donnee" du projet afin de les charger ensuite via le programme.

Premier lancement

Le menu principal vous permet de choisir entre la génération aléatoire d'un graphe ou son chargement depuis un fichier.

Une fois qu'un graphe est généré ou chargé, deux options s'offrent à vous.

Option 1: Afficher le graphe

Cette option vous permet d'afficher le graphe.

Option 2: Afficher la matrice d'adjacence

Cette option permet d'afficher la matrice d'adjacence

Option 3: Calculer la longueur d'une tournée

Cette option permet d'afficher le résultat du calcul de la longueur d'une tournée

Méthode de vérification de la connexité du graphe

J'ajoute un par un chaque sommet du graphe à une liste que j'appelle "chemin". Je parcours ensuite la liste des arêtes du graphe: pour les deux sommets reliés par chaque arête. Si le dernier sommet du chemin est égal à l'un des deux sommets de l'arête, alors supprime l'arête. Si la longueur du chemin est égal à au nombre de sommets du graphe, alors je retourne "vrai". À la fin, je retourne faux.

```
def est_connexe(graphe):
    for sommet in graphe.nodes():
        chemin = []
        chemin.append(sommet)
        graphe_temporaire = graphe.copy()

        for (sommet_a, sommet_b) in graphe_temporaire.edges():
            #
            if chemin[-1] == sommet_a:
                chemin.append(sommet_b)
                graphe_temporaire.remove_edge(sommet_a, sommet_b)

            elif chemin[-1] == sommet_b:
                chemin.append(sommet_a)
                graphe_temporaire.remove_edge(sommet_a, sommet_b)
            #
            if len(chemin) == len(graphe.nodes()):
                return True
    return False
```

Algorithme de génération aléatoire de graphe

J'instancie dans un premier temps un graphe via le modèle de données proposé par la bibliothèque python networkx. Ensuite, je génère un nombre entier aléatoirement. Celui-ci correspond au nombre de sommets qui composeront mon graphe. Ensuite, j'ajoute ou non chaque arête en passant par une valeur boléenne aléatoire.

```
def generer_graphe_aleatoire():
    graphe = nx.Graph()
    while len(graphe.nodes()) < 3:
```

```
        graphe = nx.Graph()
        random.seed()
        taille_graphe = random.randint(0, 10)
        for sommet in range(0, taille_graphe):
            graphe.add_node(str(sommet))
            for sommet_voisin in range(0, taille_graphe):
                est_vrai = bool(random.getrandbits(1))
                if est_vrai:
                    if (sommet, sommet_voisin) not in graphe.edges() and
(sommet_voisin, sommet) not in graphe.edges:
                        graphe.add_edge(str(sommet), str(sommet_voisin),
weight=random.randint(0, 100))
    return graphe
```

Algorithme de calcul des distances

Afin de calculer les distances entre les sommets, j'utilise l'algorithme de Dijkstra. Dans un premier temps j'initialise tous les sommets à "non marqué". Ensuite, j'initialise tous les labels à plus l'infini. Tant qu'il existe un sommet non marqué, je choisis le sommet y non marqué de plus petit label L. Je marque ensuite y. Pour chaque sommet z non marqué voisin de y, je calcule son nouveau label L(z) = min(L(z), L(y) + c(y, z)) où c(y, z) est le poids de l'arête qui relie y à z.

```
def calculer_matrice_adjacence(graphe):
    matrice_adjacence = []
    for sommet1 in graphe.nodes():
        liste_adjacence = []
        for sommet2 in graphe.nodes():
            liste_adjacence.append(calculer_total_poids_aretes(graphe,
dijkstra(graphe, sommet1, sommet2)))
        matrice_adjacence.append(liste_adjacence)
    return matrice_adjacence
```

Calcul de la longueur d'une tournée

Pour calculer la longueur d'une tournée, je me sers des algorithmes précédemment définis. Je calcule donc la sommes des longueurs entre chaque sommet du plus court chemin et j'ajoute la longueur entre le dernier sommet et le sommet de départ.

```
def calculer_longueur_tourner(graphe):
    return calculer_total_poids_aretes(graphe, dijkstra(graphe, graphe.nodes[0],
graphe.nodes[-1])) + calculer_total_poids_aretes(graphe, [graphe.nodes[0],
graphe.nodes[-1]])
```