Notes de cours
Semaine 23

Cours Turing

Table des matières

1 Rappel - Graphe Hamiltonien

Un parcours hamiltonien est un parcours qui passe par chaque sommet une seule fois.
Un cycle hamiltonien est un parcours hamiltonien qui commence et termine au même sommet.
Un graphe est dit hamiltonien s'il possède un cycle hamiltonien.
Un graphe est dit semi-hamiltonien s'il possède un parcours hamiltonien.

Voici un graphe hamiltonien, mais non eulérien :

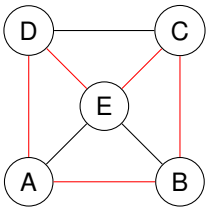

Figure 1: Graphe hamiltonien

Voici un graphe hamiltonien et eulérien:

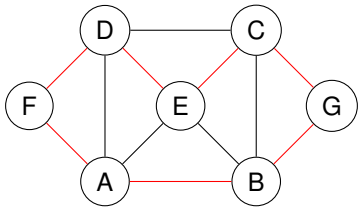

Figure 2: Graphe hamiltonien et eulérien

Un graphe complet est hamiltonien si le nombre de sommets est supérieur à 2.

Cette semaine, nous allons nous intéresser au problème du voyageur de commerce qui consiste à trouver le plus court chemin passant une seule fois par chaque ville d'un ensemble donné et revenant à son point de départ, i.e. un cycle hamiltonien.
Dans ce problème, les graphes sont généralement complets, mais ce n'est pas une condition nécessaire. Cependant, pour tous les graphes possédant n sommets, K_n est le graphe qui contiendra le plus de cycles Hamiltoniens et est donc l'occurrence la plus complexe pour le problème du voyageur de commerce

2 Le problème du voyageur de commerce

Le problème du voyageur de commerce est un problème d'optimisation qui consiste à trouver le plus court chemin passant une seule fois par chaque ville d'un ensemble donné et revenant à son point de départ.
En anglais, on parle du *Travelling Salesman Problem* (TSP).

1. une ville sera représentée par un sommet
2. la distance entre deux villes sera représentée par une arête
3. le chemin parcouru sera représenté par un cycle Hamiltonien
4. le poids du chemin parcouru sera la somme des poids des arêtes

Le problème du voyageur de commerce est un problème NP-complet, c'est-à-dire qu'il n'existe pas d'algorithme polynomial pour le résoudre. Pour être sûr de trouver la solution optimale, il faut évaluer tous les cycles hamiltoniens possibles.

Supposons que nous avons n villes et qu'il existe une route directe entre chaque paire de villes. Nous avons n choix pour la première ville, $n-1$ choix pour la deuxième ville, $n-2$ choix pour la troisième ville, etc.
Le nombre total de cycles hamiltoniens est donc $n!$, cependant certains de ces cycles sont équivalents.
Un cycle peut être parcouru dans le sens inverse sans impacter la distance totale, nous pouvons donc diviser le nombre de cycles distincts par 2.

Le point de départ n'a pas d'importance, nous pouvons donc diviser le nombre de cycles distincts par n.

Dans la figure 1, les $5 \cdot 2 = 10$ cycles hamiltoniens suivants sont équivalents :

- A-B-C-E-D-A
- B-C-E-D-A-B
- C-E-D-A-B-C
- E-D-A-B-C-E
- D-A-B-C-E-D
- A-D-E-C-B-A
- B-A-D-E-C-B
- C-B-A-D-E-C
- E-C-B-A-D-E
- D-E-C-B-A-D

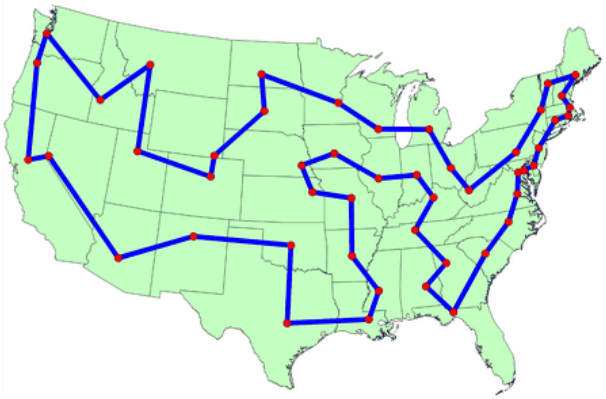

Figure 3: Le problème du voyageur de commerce appliqué aux capitales de 48 états des États-Unis.

Dans cette image, vous pouvez voir le fonctionnement d'un algorithme appliqué au problème du voyageur de commerce pour les capitales de 48 états des États-Unis.

Explosion combinatoire

Dans les figures 4, 5 et 6, nous pouvons voir le nombre de combinaisons possibles pour des tailles de problème allant de 1 à 10, 20 et 100 villes respectivement.
Le graphe de gauche nous permet de visualiser que l'ajout de la dernière ville multiplie le nombre de combinaisons possibles. Le graphe de droite, en échelle logarithmique, nous permet de voir que le nombre de combinaisons possibles augmente de manière exponentielle, car la courbe a une forme de droite.

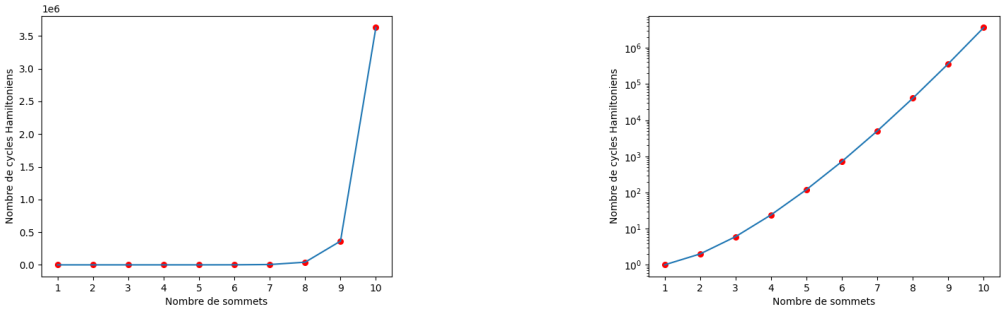

Figure 4: 10 villes

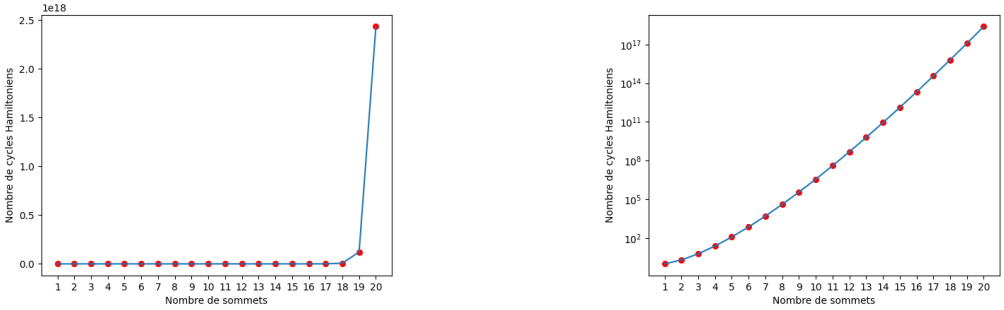

Figure 5: 20 villes

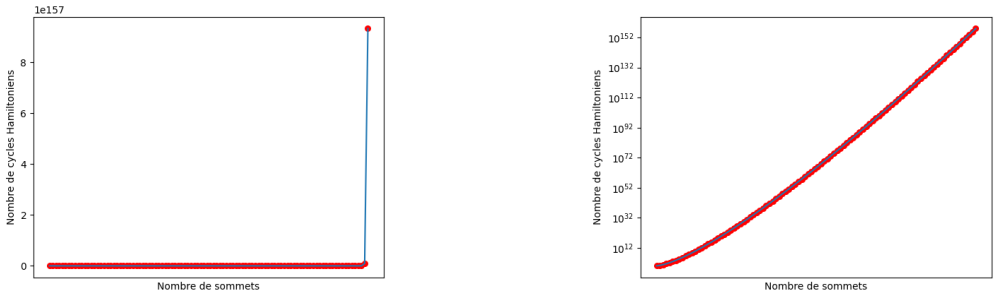

Figure 6: 100 villes

2.1 Coordonnées GPS

Pour résoudre le problème du voyageur de commerce, nous avons allons utiliser de vraies coordonnées GPS.

Les coordonnées GPS sont des coordonnées géographiques qui permettent de localiser un point sur la Terre.

Les coordonnées GPS sont exprimées en degrés, minutes et secondes. Par exemple la latitude de Lausanne est de 46.5196535 et sa longitude est de 6.6322734.

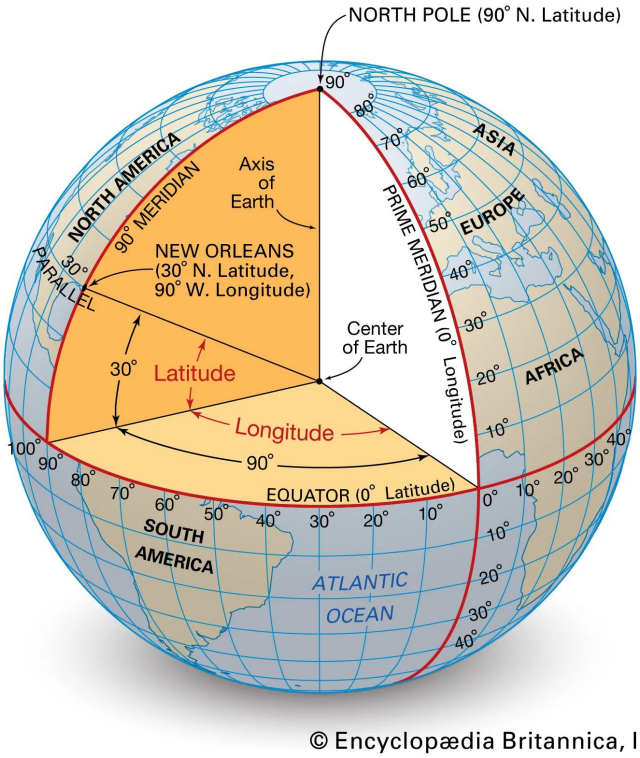

Figure 7: Représentation des coordonnées GPS sur la Terre

Un degré de latitude correspond à environ 111 km.
Un degré de longitude correspond à environ 111 km à l'équateur et diminue à mesure que l'on se rapproche des pôles.

2.2 Formule de Haversine

La formule de Haversine permet de calculer la distance entre deux points sur une sphère connaissant leur latitude et leur longitude.

Soit deux points A et B sur une sphère de rayon R:

- ϕ_A et ϕ_B sont les latitudes de A et B en radians
- λ_A et λ_B sont les longitudes de A et B en radians
- R est le rayon de la sphère (6371 kilomètres pour la Terre)

La distance d entre A et B est donnée par la formule suivante:

$$d = 2R \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_B - \phi_A}{2}\right) + \cos(\phi_A)\cos(\phi_B)\sin^2\left(\frac{\lambda_B - \lambda_A}{2}\right)}\right)$$

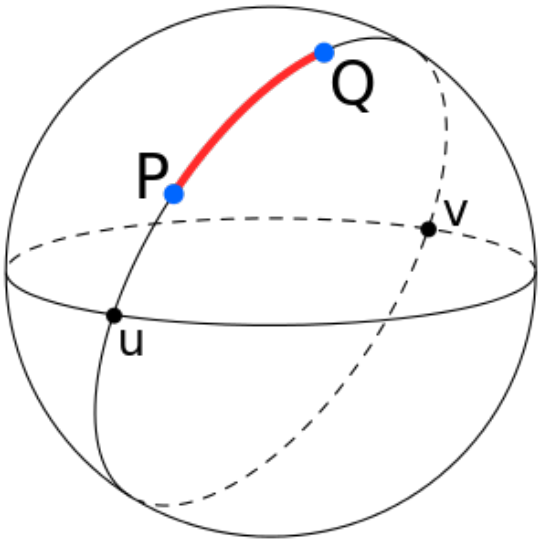

Figure 8: la formule de haversine permet de calculer la distance entre deux points sur une sphère.

2.3 Représentation en Python

2.3.1 Lecture des données

```python
# Lecture des données
def read_data(file='ch.csv'):
    cities = []
    coordinates = {}
    # Utilisation de l'encodage utf-8 pour lire le fichier, la première ligne contient les noms des colonnes, elle est ignorée
    for line in open(file, encoding="utf-8").readlines()[1:]:
        city, lat, lng = line.split(',')[0:3]
        lat = float(lat)
        lng = float(lng)
        cities.append(city)
        coordinates[city] = (lat, lng)
    return cities, coordinates
```

2.3.2 Distance de Haversine

```python
# Calcul de la distance entre deux points donnés par leur latitude et longitude
def haversine(coord1, coord2, R=6371):
    lat1, lon1 = coord1
    lat2, lon2 = coord2
    dlat = math.radians(lat2 - lat1)
    dlon = math.radians(lon2 - lon1)
    a = math.sin(dlat / 2) ** 2 + math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) * math.sin(dlon / 2) ** 2
    c = 2 * math.asin(math.sqrt(a))
    return R * c
```

2.3.3 Création de la matrice d'adjacence

```python
# Création de la matrice d'adjacence
def create_adj_matrix(cities, coordinates):
    # Initialisation de la matrice d'adjacence
    adj = [[0 for _ in range(len(cities))] for _ in range(len(cities))]
    # Remplissage de la matrice d'adjacence
    for i in range(len(cities)):
        for j in range(len(cities)):
            adj[i][j] = haversine(coordinates[cities[i]], coordinates[cities[j]])
    return adj
```

2.3.4 Coût d'une solution

```python
# Calcul du coût d'une solution
def cost(solution):
    c = 0
    for i in range(len(solution)):
        city1 = solution[i]
        city2 = solution[(i + 1) % len(solution)] # Permet de revenir à la première ville après la dernière
        idx_1 = cities.index(city1)
        idx_2 = cities.index(city2)

        w = adj[idx_1][idx_2]
        c += w
    return int(c)
```

2.3.5 Exemple d'utilisation

```
# Exemple d'utilisation
for f in ['ch.csv', 'world.csv']:

    print('Fichier', f)

    # Exemple d'utilisation
    cities, coordinates = read_data(f)
    adj = create_adj_matrix(cities, coordinates)

    # Création d'une solution aléatoire
    solution = cities.copy()
    random.shuffle(solution)

    # Visualisation de la solution
    for i in range(len(solution)):
        city1 = solution[i]
        city2 = solution[(i + 1)%len(solution)]

        idx_1 = cities.index(city1)
        idx_2 = cities.index(city2)
        plt.plot([coordinates[city1][1], coordinates[city2][1]], [coordinates[city1][0], coordinates[city2][0]],
                 color='black')

    plt.title(f'Coût du cycle Hamiltonien : {cost(solution)}')
    plt.show()
    plt.clf()
    print('    Aléatoire :', cost(solution))
```

3 Heuristiques

3.1 Approche aléatoire

L'approche aléatoire consiste à générer une solution aléatoire et à la comparer à la meilleure solution trouvée jusqu'à présent.

L'approche aléatoire est simple à implémenter mais inefficace.

L'approche aléatoire fonctionne généralement avec un nombre d'itérations fixé ou un temps d'exécution fixé.

3.2 Approche par force brute

L'approche par force brute consiste à tester toutes les combinaisons possibles pour trouver la solution optimale.

Pour n villes, il existe $\frac{(n-1)!}{2}$ cycles hamiltoniens.

Pour les données du fichier `world.csv`, il y a 234 villes, donc :

$\frac{233!}{2} \simeq 4.844 \cdot 10^{451}$

Le supercalculateur le plus puissant au monde, le Frontier, peut effectuer plus de 10^{18} opérations en virgule flottante par seconde.

Même si analyser une solution prenait seulement le temps d'une opération, il faudrait $\frac{4.844 \cdot 10^{451}}{10^{18}} = 4.844 \cdot 10^{433}$ secondes pour évaluer tous les cycles hamiltoniens.

Pour comparaison, l'âge de l'univers est d'environ $4.32 \cdot 10^{17}$ secondes (presque 14 milliards d'années.)

3.3 Algorithme du plus proche voisin

L'algorithme du plus proche voisin est un algorithme glouton qui consiste à choisir à chaque étape le sommet le plus proche du sommet actuel.

Voici les étapes de l'algorithme:

1. Choisir un sommet de départ au hasard
2. Trouver le sommet non visité le plus proche du sommet actuel
3. Ajouter ce sommet à la liste des sommets visités
4. Répéter les étapes 2 et 3 jusqu'à ce que tous les sommets aient été visités
5. Revenir au sommet de départ

3.4 2-opt

L'algorithme 2-opt est un algorithme d'optimisation qui consiste à essayer d'améliorer une solution en échangeant les arêtes par paires
Voici les étapes de l'algorithme:

1. Choisir une solution initiale
2. Pour chaque paire d'arêtes:
   (a) Calculer la distance actuelle
   (b) Inverser les arêtes
   (c) Calculer la nouvelle distance
   (d) Si la nouvelle distance est plus courte, garder la nouvelle solution
3. Si une amélioration a été trouvée, recommencer au point 2

Cet approche permettra de produire une solution où aucune paire d'arêtes ne se croise.

En supposons que nous soyons dans un espace euclidien, dans la figure 9, nous pouvons voir un exemple de deux arêtes qui se croisent dans un cycle hamiltonien.

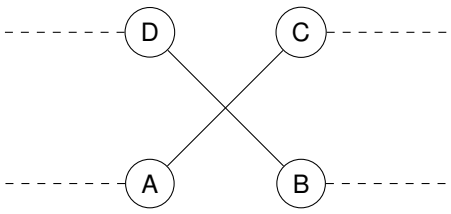

Figure 9: Exemple d'un cycle hamiltonien avec deux arêtes qui se croisent.

Peu importe le reste du cycle halmitonien dans la figure 9, il est possible de trouver une solution plus courte en échangeant les sommets B et C dans le cycle comme représenté dans la figure 10.

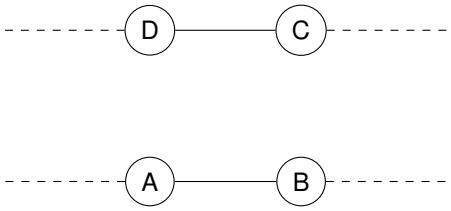

Figure 10: L'approche 2-opt permet de trouver une solution sans arêtes qui se croisent.

Cela n'est valable que dans un espace euclidien. Dans un espace non euclidien, le cycle hamiltonien optimal peut contenir des arêtes qui se croisent.

3.5 Algorithme de la colonie de fourmis

L'algorithme de la colonie de fourmis est un algorithme inspiré du comportement des fourmis.
Les fourmis laissent des phéromones sur leur chemin. Les fourmis suivantes sont attirées par les chemins avec le plus de phéromones.
L'algorithme de la colonie de fourmis est un algorithme probabiliste qui permet de trouver une solution approchée au problème du voyageur de commerce.

Voici les étapes de l'algorithme:

1. Initialiser les phéromones sur chaque arête
2. Pour chaque fourmi:
   (a) Choisir un sommet de départ
   (b) Pour chaque sommet non visité:
      i. Choisir le sommet suivant en fonction des phéromones et de la distance
      ii. Mettre à jour les phéromones
   (c) Revenir au sommet de départ
3. Mettre à jour les phéromones en fonction de la qualité des solutions trouvées
4. Répéter les étapes 2 et 3 jusqu'à ce qu'un critère d'arrêt soit atteint

Cette approche est un peu plus complexe à implémenter mais est intéressante car elle s'inspire de la nature.

Dans cette vidéo, vous pouvez voir une simulation de l'algorithme de la colonie de fourmis mais qui n'est pas appliqué au problème du voyageur de commerce.

4 A vous

Sur Moodle, vous trouverez :

- Un fichier `ch.csv` contenant les coordonnées de 178 villes de Suisse
- Un fichier `world.csv` contenant les coordonnées de 234 capitales du monde.
- Un fichier `tsp.py` contenant le code Python présenté.

Quel est le plus court cycle hamiltonien que vous arrivez à trouver dans un temps raisonnable pour chacun des 2 fichiers CSV ?

Pour référence, en utilisant uniquement les approches présentées, voici les résultats que j'ai obtenus:

- `ch.csv`: 1561 km
- `world.csv`: 170183 km