Graphes, voyageur de commerce, algorithmes génétiques

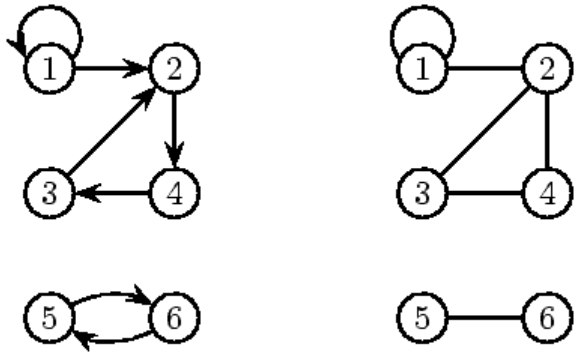
Ce TP est consacré à l'étude du parcours de graphes non orientés et à la résolution du problème du voyageur de commerce, tant de manière exacte par force brute dans des cas simples que de manière approchée à l'aide d'algorithmes génétiques dans des cas plus complexes.

1 Graphes et chemins

On appelle **graphe** la donnée d'un ensemble fini V de points (ou **sommets** — *vertices* (sing. *vertex*) en anglais) et d'un ensemble E de liens entre ces points.

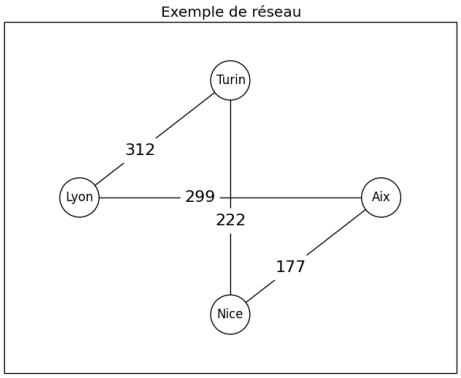
L'ensemble E de liens peut être vu comme une relation $\mathcal{R}$ sur $V \times V$. Lorsque cette relation est symétrique (c'est à dire lorsque l'existence d'un lien entre un sommet s_1 et un sommet s_2 équivaut à l'existence d'un lien entre le sommet s_2 et le sommet s_1) le graphe est dit **non orienté**. Un lien est alors appelé une **arête** (*edge* en anglais). Lorsque cette relation n'est pas symétrique, le graphe est dit **orienté**. On parle alors d'**arc** entre deux sommets.

Nous noterons généralement $\mathcal{G} = (V, E)$ un graphe. Un graphe se représente la plupart du temps par un schéma, où les sommets sont représentés par des cercles et les liens par des arcs :



On appelle **ordre** d'un graphe le nombre de ses sommets.

Un **graphe pondéré** est un graphe dont les arêtes sont affectées d'un poids, qui est en général un nombre réel. Il peut être orienté ou non. On considérera dans ce TP le seul cas où le poids affecté est strictement positif. Cela représentera par exemple des situations de distances dans un réseau routier. Voici un exemple de graphe pondéré :



Soit $\mathcal{G} = (V, E)$ un graphe. Un **chemin** $P = (S, A)$ est défini par $S = \{s_1, s_2, ..., s_k\}$, $A = \{s_0s_1, s_1s_2, ..., s_{k-1}s_k\}$ avec $S \subset V$ et $A \subset E$. Ainsi, un chemin est une suite consécutive d'arcs dans un graphe orienté. Dans le cas d'un graphe non orienté, on parle de **chaîne**.

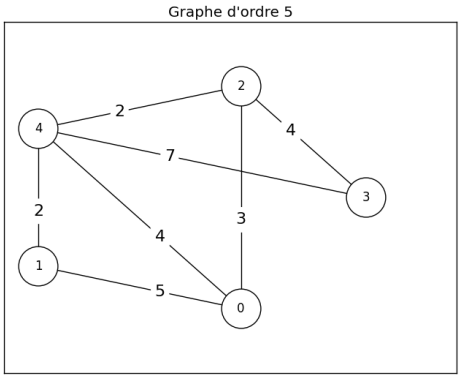
La **longueur** d'une chaîne (resp. d'un chemin) dans un graphe non pondéré est le nombre d'arêtes (resp. d'arcs) qui le constitue. Dans le cas d'un graphe pondéré, la longueur d'une chaîne (resp. d'un chemin) est la somme du poids de ses arêtes (resp. arcs).

Soit $\mathcal{G} = (V, E)$ un graphe non pondéré d'ensemble de sommets $V = \{s_1, \dots, s_{\#V}\}$. On appelle **matrice d'adjacence** $A = (a_{i,j})_{1 \leqslant i,j \leqslant \#V}$ la matrice carrée d'ordre $\#V$ dont chaque élément $a_{i,j}$ est égal à 0 si il n'y a pas de d'arête liant s_i à s_j et 1 s'il existe une arête reliant s_i à s_j. Si $\mathcal{G}$ est non orienté, sa matrice d'adjacence est symétrique.

Dans le cas d'un graphe pondéré, cette matrice est appelée **matrice des poids** : $a_{i,j}$ est égal au poids de l'arête liant s_i à s_j. S'il n'y a pas d'arête entre s_i et s_j, on convient que $a_{i,j} = +\infty$ (`math.inf` avec la version 3.5 de python et `float('inf')` avec les versions antérieures). Si le graphe est pondéré par des valeurs positives, on peut convenir que $a_{i,j} = -1$.

Dans tout le TP, on codera les matrices d'adjacence et les matrices des poids sous la forme de tableaux `numpy`.

Question 1. On considère le graphe suivant :



Construire sa matrice M des poids. On pondèrera les arêtes $[s_i s_j]$ absentes par la valeur -1 si $i \neq j$ et 0 si $i = j$.

Question 2. Construire une fonction `test_chaine(L, T)` prenant en argument la matrice des poids `T` d'un graphe pondéré et une liste `L` d'entiers, correspondant aux numéros des sommets dans $\mathcal{G}$ et qui renvoie un booléen indiquant si le parcours des sommets de `L` est possible sur $\mathcal{G}$ ou non.

Ainsi, avec le graphe ci-dessus, `test_chaine(M, [0,4,3,2]) == True` et `test_chaine(M, [1,4,3,0]) == False`.

Question 3. Écrire une fonction `longueur_chaine(L, T)` de mêmes arguments que `test_chaine(L, T)` et qui renvoie la longueur du chemin représentant `L` si il est possible et -1 sinon.

2 Présentation du problème du voyageur de commerce

On appelle **cycle** une chaîne qui a le même sommet de départ et d'arrivée.

Un graphe $\mathcal{G}$ est dit **hamiltonien** s'il possède au moins un cycle passant par tous les sommets de $\mathcal{G}$ exactement une fois ; un tel cycle est appelé **cycle hamiltonien**.

Un représentant de commerce part de sa ville d'origine et doit passer visiter ses clients dans des villes différentes, une fois et une seule. Il a bien sûr intérêt à minimiser la longueur du cycle qu'il va faire et il souhaite rentrer dans sa ville d'origine. Ce problème est appelé le « problème du voyageur de commerce » (*salesman problem*) et est apparu dans les années 1930. Avec le vocabulaire introduit plus haut, il consiste à trouver, dans un graphe hamiltonien (pondéré), un cycle hamiltonien de longueur minimale.

Un graphe (non orienté) est dit **complet** s'il existe une arête reliant tout couple de sommets. Un dénombrement immédiat montre que le graphe $\mathcal{G} = (V, E)$ est complet si, et seulement si, $\#E = \binom{\#V}{2}$. Nous nous restreindrons aux graphes complets pour la suite de ce TP. Cela revient dans notre analogie du représentant commercial à dire qu'il est possible d'aller de n'importe quelle ville à n'importe quelle autre sans faire nécessairement étape dans une ville intermédiaire déjà visitée. Par ailleurs, le graphe est non orienté, chaque chemin entre deux villes pouvant être emprunté indifféremmment dans les deux sens. Un graphe complet est bien sûr toujours hamiltonien !

Le problème du voyageur de commerce est connu pour être particulièrement difficile : il est dans la classe de complexité des problèmes **NP-complets** ; c'est une classe de problèmes algorithmiquement « très difficiles » pour lesquels nous ne savons pas écrire dans l'état actuel des connaissances informatiques d'algorithme ayant une complexité en temps polynomiale.

3 Résolution par force brute

L'approche naïve (dite par force brute) de résolution du problème du voyageur de commerce consiste à tester toutes les boucles hamiltoniennes d'origine donnée, puis à sélectionner celle qui a la plus petite longueur.

Question 4. Quelle est la complexité de cet algorithme en fonction du nombre n de sommets du graphe ?

Question 5. Écrire une fonction récursive `genere_permutations(n)` prenant en argument un entier `n` et renvoyant la liste de toutes les permutations de $[0, n-1]$, chaque permutation étant elle-même codée dans une liste. Par exemple, `genere_permutations(3)` renverra (dans cet ordre ou dans un autre) :

`[[0,1,2], [0,2,1], [1,0,2], [1,2,0], [2,0,1], [2,1,0]]`.

Question 6. Écrire une fonction `genere_boucle(n, depart)` prenant en argument un entier `n` et un entier `depart` et renvoyant une liste de toutes les boucles hamiltoniennes possibles d'origine (et d'arrivée) `depart` sur un graphe complet d'ordre n dont les sommets sont numérotés par des nombres de 0 à $n-1$. On se servira de la fonction précédente.

Question 7. Écrire une fonction `distances_boucles(LB, T)` prenant en argument une liste de boucles hamiltoniennes `LB` codées par des listes d'entiers et la matrice des poids `T` du graphe complet complet où les boucles sont situées, et qui renvoie la liste des distances associées à chacune des boucles, dans le même ordre que celui des boucles.

Question 8. Écrire une fonction `meilleure_boucle(LB, T)` de mêmes arguments que `distances_boucles` renvoyant un tuple contenant l'indice de la boucle la plus courte et son poids total.

Question 9. On dispose d'une liste de coordonnées de points repérés dans un repère orthonormé du plan affine euclidien, codée sous forme de tuples de longueur deux. Écrire une fonction `coord_vers_matrice(LC)` qui prend en argument une telle liste et renvoie une matrice des distances « à vol d'oiseau » entre les couples de points.

Question 10. On dispose des coordonnées des points suivants :

$A(0,0)$, $B(1,1)$, $C(2,4)$, $D(1,-3)$, $E(0,-5)$, $F(0,4)$, $G(-1,-5)$, $H(-2,3)$ et $I(-3,0)$.

Résoudre le problème du voyageur de commerce sur le graphe ayant ces points comme sommets, la distance étant la distance à vol d'oiseau. Le voyageur part du point A. Représenter les sommets et la solution obtenue.

4 Problème des 250 villes : création de la matrice des poids

Le problème du voyageur de commerce ne peut être résolu par force brute lorsque le nombre de villes devient relativement important (et il le devient très vite). On cherche alors des solutions approchées, accessibles en un temps raisonnable.

La recherche est encore active en ce domaine, et des défis sur des nombres de villes assez élevés ont même été organisés. Nous nous intéresserons dans la suite de ce TP au challenge du problème du voyageur de commerce pour deux cent cinquante villes tel que décrit ci-dessous :

`http://labo.algo.free.fr/defi250/defi_des_250_villes.html`.

Nous allons chercher une réponse approchée par un algorithme génétique : pour commencer nous allons créer une matrice de poids associée à ce graphe.

Question 11. On fournit un fichier externe `'villes250.txt'` contenant deux cent cinquante lignes de la forme `x, y` correspondant aux coordonnées des deux cent cinquante villes. Écrire une fonction sans argument qui renvoie une liste de deux cent cinquante tuples de la forme (x, y) correspondant à ces coordonnées. On prendra garde à ce que les coordonnées x et y soient de type numérique (flottant).

Question 12. Créer la matrice des poids associée à ce problème.

5 Problème des 250 villes : algorithme génétique

Une des méthodes pour donner une solution approchée au problème du voyageur de commerce s'appuie sur des *algorithmes génétiques*.

On part d'une génération initiale d'individus aléatoires. Un individu est une boucle hamiltonienne sur le graphe et correspond donc à une solution approchée (pas forcément performante et encore moins optimale !) du problème du voyageur de commerce.

On écrit ensuite une fonction permettant de créer la génération suivante à partir des principes suivants :

– plus un individu est performant (*i.e.* plus la route qu'il emprunte est courte), plus il a de chances de se reproduire. Un tirage au sort entre individus reproducteurs est conduit selon le principe de la roulette, principe qui sera détaillé plus loin ;

– à chaque nouvelle génération, les individus peuvent voir leur code génétique muter avec une probabilité p, qui est un paramètre (inconnu...) Une mutation consiste à sélectionner deux indices dans le code génétique d'un individu et à inverser le code génétique de cet individu entre ces deux indices. Ici, cela signifie inverser le sens du trajet entre deux villes dans la boucle hamiltonienne définissant l'individu (ce qui préserve évidemment le caractère hamiltonien du parcours).

– deux parents créent deux enfants par un système dit de *cross-over* : le premier parent donne le début (dont l'indice de fin est déterminé aléatoirement) de son parcours à un fils et le second parent donne le début de son parcours à l'autre fils. Les parcours des fils sont alors complétés par les parcours du parent dont le « code génétique » (ici, la boucle hamiltonienne qui le définit) n'a pas été utilisé, en ajoutant à la fin du code génétique du fils les sommets qui n'y apparaissent pas encore, pris dans l'ordre d'apparition dans le code du second parent ;

Question 13. La méthode `random.shuffle(L)` permet de mélanger de manière aléatoire une liste `L` passée en argument (l'action est globale). Utiliser cette méthode pour écrire une fonction `cree_initiale(N)` qui renvoie la génération initiale, codée comme liste de listes. Cette génération initiale comportera N individus, qui seront chacun représentés par une liste, commençant par l'élément 0, contenant les indices des deux cent quarante-neuf villes restantes dans un ordre aléatoire et finissant par 0.

Question 14. Écrire une fonction `meilleur_individu(population)`, qui prend en argument une liste de listes d'individus représentés par leur parcours, et qui renvoie un tuple contenant la distance totale parcourue par le meilleur individu de `population` et son parcours.

Question 15. Écrire une fonction `mutation(individu)`, qui prend en argument un individu décrit par son cycle hamiltonien et codé sous forme d'un liste et qui renvoie l'individu une fois son code génétique muté, comme décrit en début de cette partie. Les indices de mutation sont aléatoires, générés par `random.randint`.

Question 16. Écrire une fonction `crossover(p1, p2)` prenant en argument deux individus parents `p1` et `p2` et renvoyant leurs deux fils, obtenus par le procédé de *cross-over* décrit en début de cette partie.

Question 17. La roulette est une des façons de sélectionner les individus reproducteurs. Un individu donné a une probabilité proportionnelle à $f(d)$ de se reproduire où f est une fonction donnée décroissante et d la distance de parcours de l'individu.

Plus précisément, on dispose au départ d'une génération, codée sous forme de liste de listes. On calcule la liste D des distances totales parcourues par chaque individu, puis la liste F des $f(d)$. À partir de cette liste F, on crée une liste R, représentant la fonction de répartition de la variable aléatoire X, où $\mathbf{P}(X = x_k)$ est par définition la probabilité que l'individu d'indice k se reproduise. On a ainsi :

$$R_i = \frac{\displaystyle\sum_{j=0}^{i} F_j}{\displaystyle\sum_{j=0}^{N-1} F_j}.$$

On tire ensuite au hasard un flottant x entre 0 et 1. L'entier k tel que $F_k \leqslant x < F_{k+1}$ donne l'indice de l'individu choisi pour se reproduire.

Écrire une fonction `genere_roulette(population, T)`, qui prend en argument une génération et la matrice des poids T et qui renvoie la liste R décrite ici. On pourra prendre $f(d) = \dfrac{1}{(d - 0.99m)^3}$, m étant la distance parcourue par l'inividu le plus performant de la génération considérée (il s'avère empiriquement que cette fonction donne des résultats convenables).

Question 18. Écrire une fonction `indiceroulette(R)`, qui prend en argument la liste précédemment construite et qui renvoie l'indice d'un individu tiré au sort pour se reproduire.

Question 19. Créer une fonction `generation suivante` en utilisant les éléments précédents (génération par *cross-over* et mutation aléatoire).

Question 20. La tester sur plusieurs générations, en prenant soin de ne pas lancer de calculs trop longs.

Note : avec cet algorithme, en modifiant un peu la génération initiale et en ajustant un peu les paramètres, je (V. Petrov) suis arrivé après une heure de calcul à un parcours total de longueur 12.939 ; saurez-vous faire mieux ?