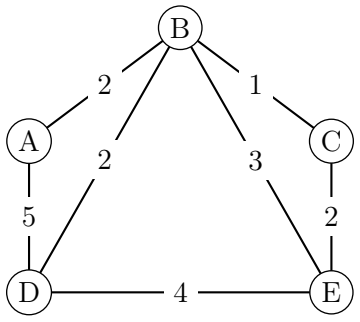
TP 2 - Graphes

Module R2.07 - BUT Informatique 1ère Année -
Contact : Frédéric Koriche (koriche@cril.fr)

Tous les TPs du module R2.07 seront programmés sous Python. Si vous utilisez votre propre machine, veillez à installer les bibliothèques nécessaires avec la dernière version de Python. Il fortement recommandé d'utiliser ***Anaconda*** *qui permet de gérer les environnements Python. Pour chaque exercice, le nombre d'étoiles indique la difficulté, allant depuis « simple » pour une étoile, jusqu'à « difficile » pour trois étoiles. Pour cette deuxième feuille de travaux pratique, l'utilisation de l'IA générative n'est pas conseillée.*

Exercice 1 ★ (*Rechercher un plus-court chemin*)
Considérons le graphe non orienté et pondéré donné dans la figure ci-dessous :



(1) A l'aide de `NetworkX`, construire le graphe non orienté pondéré correspondant à cette figure. On représentera les poids des arêtes par un attribut `"weight"`.

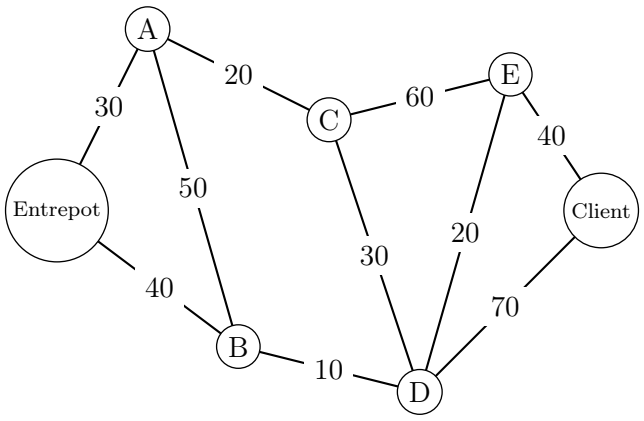
(2) A l'aide de `matplotlib`, afficher ce graphe avec les poids des arêtes. Vous pouvez utiliser le code suivant pour positionner correctement les sommets du graphe :

```
# Positionnement des sommets
pos = {
    "A": (0, 0),
    "B": (2, 1.5),
    "C": (4, 0),
    "D": (0, -2),
    "E": (4, -2),
}
```

(3) À l'aide de l'algorithme de Dijkstra (fonction de `NetworkX`), calculer un plus court chemin entre le sommet A et le sommet E, ainsi que la longueur de ce chemin.

(4) Modifier l'affichage précédent pour faire apparaître les arêtes du plus court chemin en couleur différente (par exemple en rouge et plus épaisses), les autres restant en noir.

Exercice 2 ★ (*Recherche de chemins disjoints*)
La plateforme de livraison par drones possède le réseau de corridors aériens pondéré suivant (poids = temps de vol en secondes) :



L'objectif est d'envoyer deux drones de l'entrepôt vers le client par des corridors disjoints par arête (pas de collision possible).

(1) Construire le graphe non orienté pondéré en Python avec `NetworkX`.

(2) Afficher le graphe avec les positions de la figure. A nouveau, vous pouvez utiliser le code suivant pour un positionnement des sommets qui est proche de celui de la figure :

```
# Positionnement des sommets
pos = {
    "Entrepot": (0, 0), "A": (1, 2), "B": (2, -1.5),
    "C": (3, 1), "D": (4, -2), "E": (5, 1.5),
    "Client": (6, 0)
}
```

(3) Trouver *deux chemins arête-disjoints* depuis l'entrepôt jusqu'au client. Le premier doit être de longueur minimale et le second de longueur minimale sans utiliser les arêtes du premier.

(4) Afficher les deux chemins (bleu et orange) et autres arêtes (gris).

(5) Calculer la somme des temps de vol des deux drones.

Exercice 3 ★ (*Recherche d'itinéraires sur une carte routière*)
On considère une carte routière simplifiée de la France composée de dix grandes villes. Chaque sommet représente une ville, et chaque arête représente une autoroute reliant deux villes. Chaque arête est pondérée par la distance (en kilomètres) entre les deux villes. Les villes utilisées sont données par la légende ci-dessous :

Ville	**Abréviation**
Lille	Li
Paris	Pa
Strasbourg	St
Lyon	Ly
Marseille	Ma
Nice	Ni
Nantes	Na
Rennes	Re
Bordeaux	Bo
Toulouse	To

Les distances (en km) sont données dans le tableau suivant :

Arête	Distance (km)
Pa–Li	220
Pa–St	490
Pa–Ly	465
Pa–Na	380
Na–Re	110
Na–Bo	350
Bo–To	245
To–Ma	405
Ly–Ma	315
Ly–St	490
Ly–Ni	300
Ma–Ni	200
Bo–Pa	590

Les positions géographiques des villes (longitude, latitude) sont les suivantes :

Ville	Coordonnées (lon, lat)
Li	(3.06, 50.63)
Pa	(2.35, 48.85)
St	(7.75, 48.58)
Ly	(4.84, 45.76)
Ma	(5.37, 43.30)
Ni	(7.26, 43.70)
Na	(-1.55, 47.22)
Re	(-1.68, 48.11)
Bo	(-0.58, 44.84)
To	(1.44, 43.60)

(1) Construire le graphe pondéré sous Python avec la bibliothèque `NetworkX`.

(2) Afficher le graphe avec Matplotlib. Vous utiliserez les coordonnées géographiques fournies pour positionner les sommets. Dans cet objectif, vous pouvez vous inspirer du code suivant :

```
# Tableau des positions
POSITIONS = {
    "Li": (3.06, 50.63),
    # Ecrire la suite ...
}

# Génération de la liste des positions
pos = {n: POSITIONS[n] for n in G.nodes()}

# Affichage
nx.draw(
        G,
        pos,
        # Ecrire la suite
    )
```

(3) Calculer trois itinéraires alternatifs entre Paris et Marseille. Pour cela, vous utiliserez l'algorithme de Dijkstra. À chaque nouvel itinéraire, vous supprimerez les arêtes du chemin précédent afin de forcer un trajet différent.

(4) Afficher ces trois itinéraires sur le graphe. Chaque itinéraire devra apparaître dans une couleur différente (par exemple : rouge, bleu, vert).

Exercice 4 ★ (*Routage 5G*)

Une société de télécommunications déploie un réseau 5G à Lille avec des antennes-relais aux positions stratégiques :

— **A** : Antenne centrale (Lille-Centre)

— **B** : Euralille

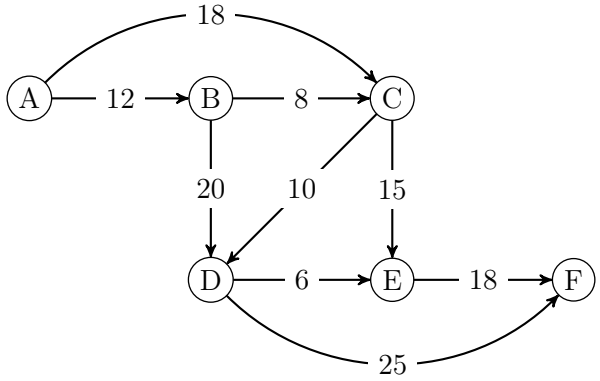
— **C** : Wazemmes

— **D** : Vauban

— **E** : Fives

— **F** : Client prioritaire (Ronchin)

Les liaisons fibre optique sont *orientées* (sens unique pour éviter les interférences) et pondérées par la *latence en millisecondes (ms)*.



(1) Construire ce graphe orienté pondéré en Python avec `NetworkX` (inutile de reproduire les positions exactes des sommets et des arcs dans cet exercice).

(2) Appliquer l'algorithme de Dijkstra depuis A et déterminer :

— La latence minimale pour atteindre F.

— Un chemin optimal $A \to F$.

— Les distances minimales vers tous les sommets.

(3) Afficher le graphe avec les arêtes du chemin optimal en rouge.

(4) Si la liaison D-F est coupée (poids ∞), quel est le nouveau chemin optimal ?

Exercice 5 ★ (*Réseaux Logistiques*)

On modélise un réseau logistique reliant plusieurs entrepôts. Chaque sommet représente un entrepôt et chaque arête dirigée représente un trajet, pondéré par un coût logistique (carburant, péages, remises). Certains trajets peuvent avoir un coût négatif, mais le réseau initial ne contient *aucun cycle négatif*. Les entrepôts sont A, B, C, D, E et les arcs pondérés du réseau sont donnés dans la table suivante :

Arcs	*Poids*
A → B	4
A → C	2
B → C	-1
B → D	2
C → B	1
C → D	4
C → E	2
D → E	-2
E → B	3

(1) Construire le graphe sous `NetworkX`.

(2) Visualiser le graphe avec `matplotlib`.
   — utiliser un layout adapté (`spring_layout` ou `kamada_kawai_layout`),
   — afficher les poids des arêtes.

(3) Appliquer l'algorithme de Bellman-Ford depuis la source A :
   — calculer les distances minimales vers tous les nœuds,
   — imprimer (sous terminal) les chemins optimaux.

(4) Visualiser les distances calculées par Bellman-Ford :
   — colorer les nœuds selon leur distance depuis A,
   — utiliser une colormap Matplotlib (par ex. `viridis`),
   — ajouter une barre de couleurs.

(5) Ajouter l'arête $E \to C$ de poids -10.
   (a) Montrer que cela crée un cycle négatif.
   (b) Relancer Bellman-Ford : que se passe-t-il ?
   (c) Quelle exception est levée par NetworkX ?

Note. L'affichage d'un gradient de couleurs pour les distances est un peu compliqué sous `Matplotlib`, car il faut récupérer la valeur de `ax`. Voici donc un exemple de code associé à la visualisation des distances. N'hésitez pas à le modifier avec d'autres colormaps.

```python
def visualiser_distances(G, distances):
    pos = nx.spring_layout(G, seed=42)
    edge_labels = nx.get_edge_attributes(G, "weight")

    # Préparation des couleurs
    dist_values = list(distances.values())
    vmin, vmax = min(dist_values), max(dist_values)
    node_colors = [distances[n] for n in G.nodes()]

    fig, ax = plt.subplots(figsize=(8, 6))

    # Dessin du graphe
    nx.draw(
        G, pos,
        with_labels=True,
        node_size=800,
        node_color=node_colors,
        cmap="viridis",
        vmin=vmin, vmax=vmax,
        ax=ax
    )
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, ax=ax)

    # Colorbar simple
    sm = plt.cm.ScalarMappable(cmap="viridis", norm=plt.Normalize(vmin=vmin, vmax=vmax))
    sm.set_array([])  # nécessaire pour Matplotlib
    fig.colorbar(sm, ax=ax, label="Distance depuis la source")

    plt.title("Distances calculées par Bellman–Ford")
    plt.show()
```

Exercice 6 ★★ (*Graphe d'un réseau social*)

Les réseaux sociaux peuvent être modélisés par des graphes non orientés (ou orientés) et pondérés. Les sommets représentent les personnes et les arêtes représentent les liens entre personnes. Le poids d'une arête peut modéliser la distance sociale entre deux personnes, fondée par exemple sur le nombre d'interactions journalières (messages) envoyées entre elles. Si $f(u, v)$ décrit la fréquence des messages journaliers échangés entre u et v, alors le poids correspondant est donné par

$$w(u, v) = \frac{1}{1 + f(u, v)}$$

Le poids est donc d'autant plus petit que la fréquence d'échanges (et donc la distance sociale) est grande. Sous `NetworkX`, un réseau social peut être simulé par un graphe de *Watts–Strogatz* non orienté et connexe, où n est le nombre de personnes (sommets), k est le degré moyen par sommet et p la probabilité de reconnecter (*rewire*) une arête entre deux personnes. Le but est de découvrir quelle sont les personnes socialement les plus proches d'une personne donnée.

(1) Ecrire une fonction `construire_graphe` qui retourne un graphe de Watts–Strogatz connecté et pondéré (fonction `connected_watts_strogatz_graph`) avec par défaut n =

10, $k = 4$ et $p = 0.3$. La fréquence $f(u, v)$ de messages journaliers échangés entre deux personnes est un entier aléatoire entre 0 et 9.

(2) Ecrire une fonction `afficher_graphe` qui affiche le graphe G sous `Matplotlib`. Comme les poids sont des réels, il est important de les tronquer pour visualiser le graphe correctement. Pour cela, vous pouvez utiliser la portion de code suivante :

```
# Tronquer les poids à 2 décimales
edge_labels = {
    (u, v): f"{w:.2f}" for (u, v), w in nx.get_edge_attributes(G, 'weight').items()
}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
```

(3) Ecrire une fonction `distances_bellman_ford` qui prend en entrée le graphe G et une personne `source` et retourne en sortie le dictionnaire des distances des plus courts chemins depuis la source jusqu'aux autres sommets connectés à cette source.

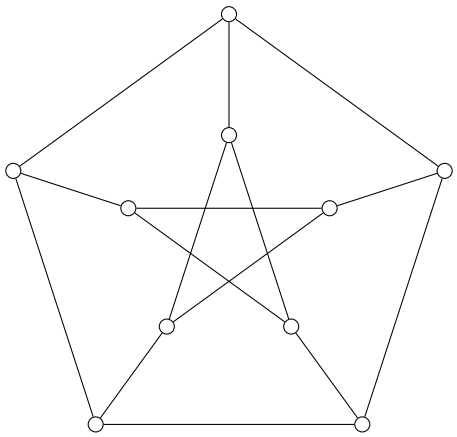
(4) Ecrire une fonction `k_plus_proches` qui prend donc en entrée une personne `source`, le dictionnaire `distances` construit précédemment et un entier k, et retourne en sortie un dictionnaire des distances des k personnes les plus proches de `source`. *Astuce :* trier le dictionnaire par distances croissantes, enlever la source et retourner le dictionnaire formé par les k premières entrées.

(5) Ecrire une dernière fonction `afficher_k_proches` qui prend en entrée le graphe G, une personne `source`, un dictionnaire `closest_k` des k personnes socialement les plus proches de `source`, et qui affiche avec des couleurs distinctes le sommet `source`, les sommets de `closest_k` et tous les autres sommets du graphe.

(6) Tester votre programme sur divers sommets du graphe.

Exercice 7 ★ (*Graphe de Petersen*)

Le graphe de Petersen est un petit bijou de la théorie des graphes : compact, très symétrique, mais suffisamment « tordu » pour servir d'exemple contre-intuitif dans de nombreux théorèmes. Il est décrit dans la figure suivante :



Nous allons analyser quelques statistiques de ce graphe.

(1) En vous aidant de la documentation de `NetworkX`, construire une première fonction retournant le graphe de Petersen.

(2) Ecrire une fonction permettant d'afficher ce graphe avec les sommets en bleu.

(3) En utilisant l'algorithme de Floyd-Warhsall, écrire un algorithme calculant la matrice des distances entre chaque paire de sommets du graphe de Petersen.

(4) A partir de la matrice des distances, écrire une fonction retournant les excentricités de chaque sommet.

(5) A partir des excentricités, écrire un fonction calculant le rayon et le diamètre du graphe.

(6) Ecrire ensuite une fonction retournant tous les centres du graphe.

(7) Ecrire enfin une fonction affichant avec une couleur distincte les centres du graphe.

(8) Démontrer par un argument simple que, pour le graphe de Petersen, tous les sommets sont des centres.

Exercice 8 ★ (*Géométrie des graphes*)
Contrairement au graphe de Petersen, la plupart des graphes non orientés ont des propriétés géométriques (rayon, diamètre, centres) qui ne sont pas homogènes. L'objectif de cet exercice est d'utiliser les fonctions de l'exercice précédent pour obtenir des statistiques sur différentes classes de graphes.

(1) Calculer le rayon, le diamètre et les centres des graphes suivants :

Graphe	Fonction
Krackhardt Kite	`krackhardt_kite_graph`
Moebius Kantor	`moebius_kantor_graph`
Sedgewick Maze	`sedgewick_maze`
Tétrahédrique	`tetrahedral_graph`

Expliquer pourquoi dans certains cas tous les sommets sont des centres.

(2) En utilisant les graphes d'Erdos avec 10 sommets (fonction `erdos_renyi_graph`) et en faisant varier la probabilité d'arêtes p de 0.1 à 0.9 (par tranches de 0.1) déterminer si le graphe généré est connexe, et dans ce cas calculer son rayon, son diamètre et ses centres. Le nombre de centres augmente-t-il systématiquement quand p augmente ?

Exercice 9 ★★ (*Dijkstra vs. Bellman-Ford : Comparaison statistique*)
L'objectif de cet exercice est de comparer expérimentalement les performances des algorithmes de Dijkstra et de Bellman–Ford pour le calcul de plus courts chemins depuis une source, sur une famille de graphes aléatoires contrôlés. Les graphes considérés sont construits en deux étapes :

1. génération d'un arbre aléatoire à n sommets ;
2. ajout d'arêtes supplémentaires avec une probabilité p pour chaque paire de sommets non adjacents.

Les poids des arêtes sont tirés uniformément dans l'intervalle $[1, 10]$. Par construction, les graphes sont connexes. Quatre expériences seront menées, correspondant à quatre régimes de densité :

$$p \in \{0.002,\ 0.05,\ 0.3,\ 0.7\}.$$

Intuivement, le régime $p \leq 0.002$ correspond à des graphes très clairsemés (presque des arbres), tandis que le régime $p \geq 0.700$ correspond à des graphes denses (presque complets). Pour chaque valeur de p, on génère 100 graphes aléatoires à $n = 100$ sommets, et on mesure les temps d'exécution moyens et écarts-types des deux algorithmes. L'exercice est découpé en questions simples, chacune associée à une fonction Python.

(1) *Génération d'un arbre aléatoire.* Écrire une fonction Python `generate_random_tree(n)` qui génère un arbre aléatoire à n sommets à l'aide de `networkx.random_tree`, puis attribue à chaque arête un poids entier tiré uniformément dans $[1, 10]$. La fonction retourne l'arbre ainsi généré.

(2) *Ajout d'arêtes aléatoires.* Écrire une fonction `add_random_edges(G, p)` qui parcourt toutes les paires de sommets (u, v) non adjacents dans G et ajoute l'arête (u, v) avec probabilité p, avec un poids tiré uniformément dans $[1, 10]$. La fonction retourne le graphe ainsi généré. Notons qu'il est non orienté, pondéré et connexe.

(3) *Implémentation des algorithmes.* Écrire deux fonctions `dijkstra(G, source)` et `bellman_ford(G, source)` qui appellent respectivement `nx.single_source_dijkstra_path_length` et `nx.single_source_bellman_ford_path_length`. Les deux fonctions renvoient le dictionnaire des distances depuis la source.

(4) *Mesure du temps d'exécution.* Écrire une fonction `measure_runtime(algorithm, G, source, repeats=3)` qui mesure le temps d'exécution moyen de l'algorithme donné sur le graphe G, en répétant l'appel `repeats` fois. Utiliser `time.perf_counter()` (qui est plus précis que `time.time` pour mesurer des temps de calcul). La fonction renvoie la moyenne et l'écart-type des temps mesurés.

(5) *Expérience pour une valeur de p.* Écrire une fonction `run_experiment(n, p, n_graphs)` qui :

   (a) génère *n_graphs* graphes aléatoires selon les questions 1 et 2 ;

   (b) mesure les temps d'exécution de Dijkstra et Bellman-Ford depuis la source 0 ;

   (c) calcule :

      — le nombre moyen d'arêtes,

      — la moyenne et l'écart-type des temps pour chaque algorithme.

   La fonction renvoie un dictionnaire contenant ces informations.

(6) *Lancement des quatre expériences.* Écrire une fonction `run_all_experiments()` qui exécute `run_experiment(100, p, 100)` pour les quatre valeurs :

$$p = 0.002,\ 0.05,\ 0.3,\ 0.7.$$

   La fonction renvoie la liste des résultats.

(7) *Visualisation des résultats.* Écrire une fonction `plot_results(results)` qui trace, en fonction de p :

   — le temps moyen d'exécution de Dijkstra ;

   — le temps moyen d'exécution de Bellman–Ford.

   Utiliser `matplotlib`. Vous pouvez vous inspirer du code suivant :

```python
def plot_results(results):
    ps = [r["p"] for r in results]
    dijkstra_times = [r["dijkstra_mean"] for r in results]
    bf_times = [r["bf_mean"] for r in results]

    plt.figure(figsize=(10, 6))
    plt.plot(ps, dijkstra_times, marker='o', label="Dijkstra")
    plt.plot(ps, bf_times, marker='o', label="Bellman–Ford")
    plt.xlabel("Probabilité p")
    plt.ylabel("Temps moyen (s)")
    plt.title("Comparaison des temps d'exécution")
    plt.legend()
    plt.grid(True)
    plt.show()
```

(8) A partir de vos expérimentations, répondez aux questions suivantes et discutez-en avec votre responsable de TP.
   — Pourquoi Bellman–Ford devient très lent lorsque p augmente ?
   — Pourquoi Dijkstra reste performant même pour des graphes denses ?

Exercice 10 ★★★ (*Graphes de mots*)

La *distance d'édition* (appelée aussi *distance de Levenshtein*) entre deux mots d'un langage a et b est le nombre minimal d'opérations nécessaires pour transformer a en b. Les opérations autorisées sont :

— *insertion* d'un caractère ;
— *suppression* d'un caractère ;
— *substitution* d'un caractère par un autre.

Par exemple : `table` ⟶ `sable` peut être obtenu par une seule substitution (`t` remplacé par `s`), donc : $d(\texttt{table}, \texttt{sable}) = 1$.

La distance d'édition est une véritable distance : elle est toujours positive, symétrique, et vérifie l'inégalité triangulaire. Elle est très utilisée en traitement automatique des langues, en bio-informatique (comparaison de séquences ADN), et dans les algorithmes de correction orthographique.

Dans cet exercice, on considère un ensemble de mots français de 4 à 6 lettres. Chaque mot constitue un sommet d'un graphe non orienté. Deux mots sont adjacents si leur distance d'édition est inférieure ou égale à un entier k. L'objectif est de :

— construire un graphe de mots pour un ensemble donné ;
— déterminer le plus petit k pour lequel le graphe devient connexe ;
— calculer la matrice des distances (Floyd–Warshall) ;
— déterminer le rayon, le diamètre et les centres du graphe ;
— afficher le graphe et mettre en évidence ses centres.

(1) Écrire une fonction Python `levenshtein(a,b)` calculant la distance d'édition entre deux mots a et b. C'est la partie la plus difficile de l'exercice, donc vous pouvez vous inspirer d'algorithmes sur le Web pour écrire votre code (note : vous pouvez par exemple utiliser en import le package `itertools`).

(2) Écrire une fonction `construire_graphe(mots, k)` qui :
   — crée un graphe non orienté dont les sommets sont les mots ;
   — ajoute une arête entre deux mots si leur distance d'édition est $\leq k$.

(3) Écrire une fonction `afficher_graphe(G)` utilisant `networkx` et `matplotlib` pour afficher le graphe.

(4) Écrire une fonction `matrice_distances(G)` calculant la matrice des distances du graphe à l'aide de l'algorithme de Floyd–Warshall.

(5) Écrire une fonction `rayon_diametre_centres(G)` qui calcule :
   — le rayon du graphe ;
   — le diamètre du graphe ;
   — l'ensemble des centres (sommets d'excentricité minimale).

(6) Écrire une fonction `plus_petit_k_connexe(mots)` qui :
   — construit le graphe pour $k = 0, 1, 2, \ldots$;
   — renvoie le plus petit k pour lequel le graphe devient connexe ;
   — renvoie également le graphe correspondant.

(7) Écrire une fonction `afficher_centres(G, centres)` qui affiche le graphe en coloriant les centres dans une couleur distincte.

(8) Écrire une fonction principale de la forme suivante :

```
mots = [ ... ] # Corpus à remplir

k_min, G_connexe = plus_petit_k_connexe(mots)
print("Plus petit k rendant le graphe connexe :", k_min)

afficher_graphe(G_connexe, title=f"Graphe connexe obtenu pour k = {k_min}")

rayon, diametre, centres = rayon_diametre_centres(G_connexe)
print("\nRayon :", rayon)
print("Diamètre :", diametre)
print("Centres :", centres)

afficher_centres(G_connexe, centres)
```

(9) Tester le programme sur le corpus suivant :

```
mots = [
  "lune", "vent", "bois", "chat", "jour",
  "arbre", "table", "sable", "cadre", "sucre",
  "vivre", "libre", "fibre", "soleil", "chemin",
  "poulet", "terreau", "plume", "route", "ombre"
  ]
```

(10) Tester le programme sur le corpus suivant :

```
mots = [
  "pomme", "gomme", "somme", "tombe",
  "table", "sable", "fable", "cable",
  "chien", "chier", "choir", "choix",
  "ligne", "signe", "digne"
]
```

(11) Comparer les résultats pour ces deux corpus et discutez de vos observations avec le responsable de TP.

Exercice 11 ★★★ (*Métro Parisien*)
Passons à un réseau réel, celui du métro Parisien, dont la table de données est fournie dans le fichier `metro_rer_idf.csv`. Le code `tp2_exercice11.py` définit les fonctions principales permettant de construire le graphe à partir du fichier et d'afficher ce graphe. Il s'agit ici d'un multigraphe orienté et pondéré, dont les sommets sont les stations et les poids sont les temps de trajet (en minutes) entre deux stations adjacentes. À partir de ce code :

(1) Construire une fonction `voisins_sortants` qui prend en entrée le graphe G et la chaîne de caractères d'une station `source` et retourne en sortie toutes les stations qui sont voisins sortants de cette station. Tester cette fonction sur la station « Nation ».

(2) Construire une fonction `plus_court_chemin` qui prend en entrée le graphe G, une station `source` et une station `target`. La fonction doit retourner une erreur si la source ou la cible ne sont pas dans le graphe. Si ces deux stations sont bien dans le graphe, alors la fonction doit retourner un plus court chemin depuis `source` jusqu'à `target` en utilisant l'algorithme de Dijkstra. Tester la fonction avec « Nation » pour la source et « Auber » pour la cible.

(3) Construire une fonction `affiche_chemin` qui prend en entrée le graphe G et un chemin `chemin` et affiche en sortie le graphe avec le chemin. Choisissez des couleurs distinctes. Par exemple, les arêtes sont en gris, les sommets en rouge, les stations du chemin en bleu, la station de départ en vert, la station d'arrivée en orange, et toutes les arêtes du chemin en bleu épais.

(4) Construire une fonction `k_stations_les_plus_proches` qui prend en entrée le graphe G, une station `source` et un entier k. La fonction doit retourner une erreur si la source n'est pas dans le graphe. Si la source appartient bien au graphe, alors la fonction doit retourner les k stations les plus proches de la source (sans inclure celle-ci). Tester la fonction avec « Nation ». *Astuce :* utiliser l'algorithme de Bellman-Ford qui retourne le dictionnaire des distances depuis la source, enlever du dictionnaire la source, trier le dictionnaire par ordre de distances croissantes, et récupérer les k premiers éléments du dictionnaire.

(5) Construire une fonction `affiche_stations_les_plus_proches` qui prend en entrée le graphe G, une station `source` et un dictionnaire `proches` construit par la fonction précédente. La fonction doit afficher par des couleurs distinctes la station source et toutes les stations de `proches`. Par exemple, utiliser l'orange pour la source, le vert pour les stations proches, et le rouge pour toutes les autres stations du graphe.

(6) Construire une fonction `statistiques_graphe` qui construit d'abord le graphe H non orienté sous-jacent à G. La fonction retourne alors le rayon, le diamètre et les centres de H.

(7) Construire une dernière fonction `affiche_centres` qui prend en entrée le graphe G et une liste de stations `centres` et affiche avec une couleur distincte ces centres.

Exercice 12 ★★★ (*Dijkstra versus A^* : Comparaison statistique*)
L'objectif de ce projet est de comparer expérimentalement les performances de trois algorithmes de plus court chemin sur des labyrinthes représentés sous forme de grilles :

— Dijkstra (noté D),

— A*avec heuristique de Manhattan (noté AM),

— A*avec heuristique euclidienne (noté AE).

Les labyrinthes sont modélisés comme des graphes pondérés, dont les sommets sont les cases d'une grille $n \times n$, et les arêtes relient les cases adjacentes. Le sommet de départ est $(0, 0)$ (en haut à gauche) et le sommet d'arrivée est $(n-1, n-1)$ (en bas à droite). Le code Python permettant de générer les quatre types de labyrinthes suivants vous est fourni dans le fichier `tp2_exercice12.py`. Il contient les générateurs des labyrinthes suivants :

1. **Labyrinthe parfait** (connexe, sans cycles).
2. **Labyrinthe dense** (labyrinthe parfait auquel on ajoute des cycles).
3. **Labyrinthe avec obstacles** (suppression contrôlée d'arêtes tout en garantissant la connectivité).
4. **Labyrinthe pondéré** (grille complète avec poids aléatoires sur les arêtes).

Chaque fonction renvoie un graphe NetworkX connexe dont les sommets sont les couples (i, j). En plus de la génération des labyrinthes, le code contient déjà la fonction `visualiser_labyrinthe(G, n)` qui affiche le labyrinthe sous forme de grille.

A partir de ce code pré-rempli, vous devez implémenter les fonctions suivantes.

(1) *Heuristiques pour A^*.* Implémentez les deux heuristiques suivantes :

— **Heuristique de Manhattan** :

$$h_{\text{Manhattan}}((x, y), (x', y')) = |x - x'| + |y - y'|.$$

— **Heuristique euclidienne** :

$$h_{\text{Euclidienne}}((x, y), (x', y')) = \sqrt{(x - x')^2 + (y - y')^2}.$$

Ces fonctions seront passées en paramètre à `networkx.astar_path_length`.

(2) *Fonctions de mesure du temps.* Vous devez implémenter trois fonctions :

— `dijkstra_temps(G, start, goal)`

— `astar_temps(G, start, goal, heuristique)`

— `tester_labyrinthe(G, start, goal)`

Chaque fonction doit :

— appeler l'algorithme correspondant de `NetworkX` ;

— mesurer le temps d'exécution avec `time.perf_counter()` ;

— retourner le coût du chemin et le temps d'exécution.

La fonction `tester_labyrinthe` doit en plus calculer un taux d'erreur pour AM et AE. Pour chaque graphe, on compte :

— erreur = 1 si le coût trouvé par A*est strictement supérieur à celui de Dijkstra ;

— erreur = 0 sinon.

Le taux d'erreur est donc le nombre moyen d'erreurs faites par A*.

(3) *Fonction d'expérimentation.* Implémentez la fonction :

```
experimenter_type(type_fonction, n, repetitions)
```

Cette fonction doit :

- générer `repetitions` labyrinthes du type donné ;
- exécuter D, AM et AE sur chacun ;
- enregistrer les temps d'exécution et les erreurs ;
- retourner un dictionnaire contenant :
  - le temps moyen et l'écart-type pour D, AM et AE ;
  - le taux d'erreur pour AM et AE.

(4) *Visualisation des performances.* En utilisant la documentation de `matplotlib`, implémentez la fonction :

```
visualiser_performances(resultats, titre)
```

Elle doit afficher un histogramme comparant les temps moyens des trois algorithmes.

(5) *Visualisation des erreurs.* En utilisant la documentation de `matplotlib`, implémentez la fonction :

```
visualiser_erreurs(resultats, titre)
```

Elle doit afficher un histogramme comparant le taux moyen d'erreurs des deux algorithmes AE et AM.

(6) *Fonction principale.* Votre fonction `main()` doit :

- lancer les expériences pour les quatre types de labyrinthes ;
- afficher les résultats ;
- produire les histogrammes correspondants.

La fonction est déjà pré-remplie dans le code fourni, mais vous pouvez la modifier à votre convenance.

A partir de vos expérimentations, répondez aux questions suivantes :

- Quel algorithme est le plus rapide en moyenne ?
- L'heuristique de Manhattan est-elle plus efficace que l'heuristique euclidienne ?
- Le taux d'erreur de A*est-il significatif ?
- Comment la structure du labyrinthe influence-t-elle les performances ?

Discutez de vos observations avec le responsable de TP.