Exercice 1 (6 points)

Cet exercice porte sur les graphes et la programmation orientée objet.

Certains sportifs sont adeptes des *courses d'orientation* : munis d'une fiche de contrôle, d'une carte et d'une boussole, les participants doivent trouver leur itinéraire dans le but de relier des points de contrôle, appelés balises. Chaque balise est numérotée et équipée d'un poinçon permettant de l'identifier : lorsqu'un participant trouve une balise, il poinçonne sa fiche de contrôle avec ce poinçon, différent d'une balise à l'autre.

Le site *La forêt des chênes* propose trois itinéraires de courses d'orientation, de niveaux différents : un itinéraire vert de niveau facile, un itinéraire rouge de niveau moyen, et un itinéraire noir de niveau difficile.

Partie A

Le site *La forêt des chênes* est représenté par le graphe ci-dessous dans lequel chaque nœud correspond à une balise et chaque arête représente un chemin reliant deux balises. Pour faire référence aux couleurs des itinéraires, on utilise des symboles avec la correspondance suivante :

- un triangle pour indiquer que l'itinéraire vert passe par cette balise ;
- un carré pour indiquer que l'itinéraire rouge passe par cette balise ;
- un pentagone pour indiquer que l'itinéraire noir passe par cette balise.

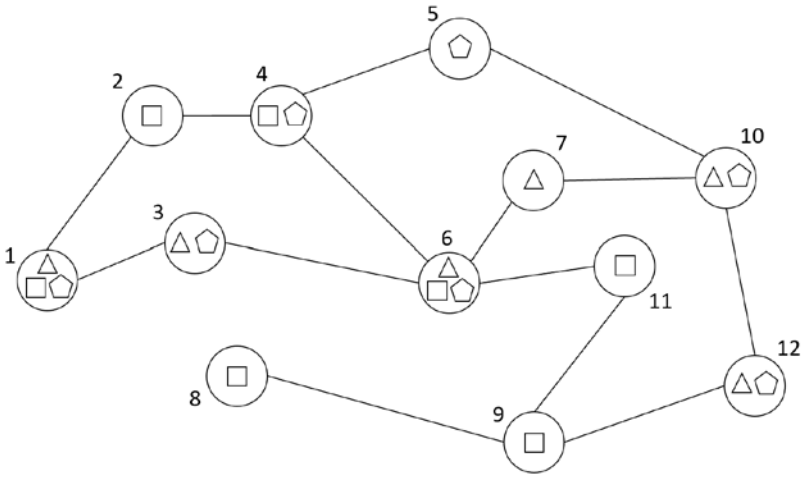

Figure 1. Graphe représentant le site *La forêt des chênes*

Une balise peut donc faire partie d'un ou de plusieurs itinéraires.

Le point de départ est commun aux trois itinéraires : il s'agit de la balise portant le numéro 1. On peut alors constater, par lecture du graphe, que l'itinéraire vert et

l'itinéraire noir se terminent à la balise numéro 12, tandis que l'itinéraire rouge se termine à la balise numéro 8.

On choisit de modéliser ce site en Python en utilisant une classe `Balise` dont le code est donné en **Annexe**. Cette **Annexe** n'est pas à rendre avec la copie.

Chaque balise du site est représentée par une instance de la classe `Balise` et a pour attributs :

- `num_balise`, un nombre entier correspondant au numéro de la balise ;
- `couleurs_balise`, une liste de chaîne de caractères dont chaque élément est une couleur de la balise ;
- `voisines`, une liste d'instances de la classe `Balise` ;
- `visitee`, un indicateur booléen qui indique si la balise a déjà été poinçonnée ou non (cet attribut utilisé dans la partie B).

1. Recopier et compléter le code de la ligne 28 afin d'instancier la balise portant le numéro 12.
2. Écrire l'instruction de la ligne 38 permettant de compléter l'implémentation du graphe de la Figure 1.
3. Dans la console, on saisit l'instruction suivante :

   ```
   >>> balise4.methode1()
   ```

   Indiquer le résultat renvoyé à l'écran.
4. Dans la console, on saisit la suite d'instructions suivante :

   ```
   >>> balise4.methode2('rouge')
   >>> balise4.methode3('vert')
   >>> balise4.couleurs_balise
   ```

   Indiquer le résultat renvoyé à l'écran.

On instancie à nouveau la balise numéro 4 pour revenir au graphe représenté sur la Figure 1.

Ci-après, on donne le code incomplet d'une fonction `itineraire`.

Cette fonction renvoie, sous la forme d'une liste, les numéros des balises qui correspondent à l'itinéraire reliant `balise_debut` et `balise_fin`, et dont le niveau de difficulté (vert, rouge ou noir) est donné par le paramètre `couleur`.

Par exemple, l'appel `itineraire(balise1, balise12, 'vert')` renvoie la liste d'entiers `[1, 3, 6, 7, 10, 12]`.

```
1 def itineraire(balise_debut, balise_fin, couleur):
2     assert couleur in balise_debut.couleurs_balise
```

```
 3      assert couleur in balise_fin.couleurs_balise
 4      balise = balise_debut
 5      chemin = [balise]
 6      while balise.num_balise != ...:
 7          for b in balise.voisines:
 8              if (couleur in ...) and (b not in ...):
 9                  balise = ...
10                  chemin.append(balise)
11      return [b.num_balise for b in chemin]
```

5. Recopier et compléter les lignes 4 à 8 de la fonction `itineraire`.

Le site *La forêt des chênes* décide de définir une nouvelle épreuve en récompensant les participants qui reviennent avec l'ensemble des balises poinçonnées sur leur fiche de contrôle. Les couleurs des balises ne sont donc pas prises en compte dans cette épreuve.

On rappelle que le point de départ de tous les itinéraires est la balise portant le numéro 1. Un participant décide ainsi, pour remplir cet objectif, d'appliquer un algorithme de parcours en profondeur du graphe, en choisissant la balise de plus petit numéro lorsqu'il y a plusieurs choix possibles.

6. Donner, dans l'ordre, les numéros des balises rencontrées par ce participant.

Partie B

Dans cette partie, les couleurs des balises ne sont plus prises en compte.

Le site *La forêt des chênes* est également recommandé par les amateurs de trails (courses à pied en milieu naturel) à la recherche de performances. Ainsi, sur chaque arête du graphe ci-dessous, on mentionne le temps moyen, en minute, permettant de relier deux balises voisines, les balises étant reliées par des chemins plats.

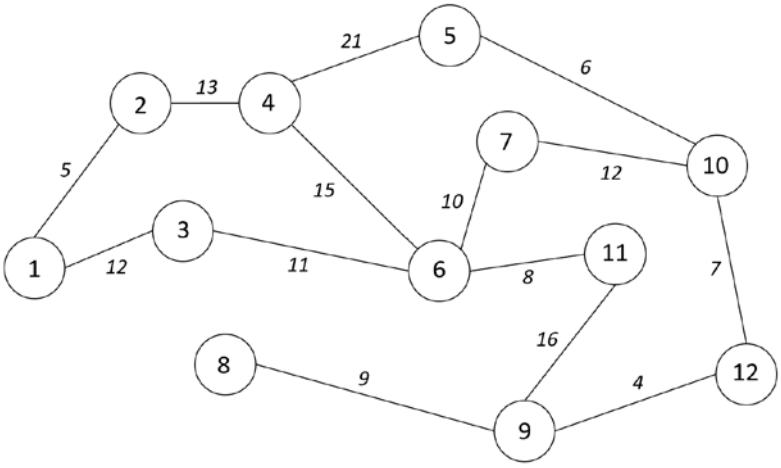

Figure 2. Graphe représentant le site *La forêt des chênes* avec indication des temps moyens en minute

On modifie alors l'implémentation du graphe en Python en remplaçant la liste des balises voisines par une liste de tuples indiquant la balise voisine et le temps moyen pour la rejoindre.

Par exemple, le code de la ligne 30 concernant la balise 1 est remplacé par : `balise1.voisines = [(balise2, 5), (balise3, 12)]`.

7. Donner le code qui permet, de la même façon, de définir les balises voisines de la balise portant le numéro 4 et qui permet donc de remplacer l'instruction de la ligne 33 dans le code donné en **Annexe**.

On donne ci-dessous le code d'une fonction Python `mystere` prenant pour paramètre `balise`, une instance de la classe `Balise`.

```
1 def mystere(balise):
2     meilleure_balise = None
3     mini = -1
4     for b, t in balise.voisines:
5         if (b.visitee == False) and (mini == -1 or t < mini):
6            meilleure_balise, mini = b, t
7     return meilleure_balise
```

Dans la question suivante, on suppose que la valeur de l'attribut `visitee` de toutes les instances de la classe `Balise` est `False`.

8. Indiquer le résultat renvoyé à l'écran lorsque l'utilisateur saisit, dans la console, l'instruction suivante :

```
>>> mystere(balise10).num_balise
```

Dans la suite de l'exercice, on s'intéresse à un sportif qui, pour déterminer son itinéraire, choisit, à chaque balise, de s'orienter vers la balise voisine la plus proche en temps, sans jamais repasser par une balise déjà visitée.

Ce sportif part de la balise numéro 1 pour rejoindre la balise numéro 12.

9. Donner, dans l'ordre, les numéros des balises rencontrées par ce sportif.

On donne ci-dessous le script, incomplet, d'une fonction `itineraire_trail` prenant pour paramètres `balise_debut` et `balise_fin`, deux instances de la classe `Balise`.

Cette fonction doit renvoyer, sous la forme d'une liste d'entiers, les numéros des balises rencontrées successivement par le sportif lorsqu'il part de `balise_debut` pour rejoindre `balise_fin`. Si un tel itinéraire ne peut pas être déterminé, la fonction doit renvoyer `None`.

On supposera qu'avant chaque appel à la fonction `itineraire_trail`, l'utilisateur aura exécuté les instructions permettant d'attribuer la valeur `False` à l'attribut `visitee` de toutes les instances de la classe `Balise`.

```
 1 def itineraire_trail(balise_debut, balise_fin):
 2     balise_debut.visitee = True
 3     balise = balise_debut
 4     chemin = [balise]
 5     while balise_fin not in chemin:
 6         prochaine = ...
 7         if prochaine != None:
 8             ...
 9         else:
10             return None
11     return [b.num_balise for b in chemin]
```

10. Recopier les lignes 5 à 9 en complétant la ligne 6 et en ajoutant autant de lignes que nécessaires à la place de la ligne 8 afin de compléter la fonction `itineraire_trail`.

11. Indiquer, sans justifier, à quel type d'algorithme appartient le script précédent parmi les trois propositions suivantes :

    - **Proposition A** : algorithme glouton ;
    - **Proposition B** : algorithme de recherche par force brute ;
    - **Proposition C** : algorithme des k plus proches voisins.

12. Donner un avantage et un inconvénient de ce type d'algorithme.

Annexe

```
 1 class Balise:
 2     def __init__(self, numero, couleurs):
 3         self.num_balise = numero
 4         self.couleurs_balise = couleurs
 5         self.voisines = []
 6         self.visitee = False
 7
 8     def methode1(self):
 9         return [b.num_balise for b in self.voisines]
10
11     def methode2(self, couleur):
12         self.couleurs_balise = [c for c in
self.couleurs_balise if c != couleur]
13
14     def methode3(self, couleur):
15         self.couleurs_balise.append(couleur)
16
17 balise1 = Balise(1, ['vert', 'rouge', 'noir'])
18 balise2 = Balise(2, ['rouge'])
19 balise3 = Balise(3, ['vert', 'noir'])
```

```
balise4 = Balise(4, ['rouge', 'noir'])
balise5 = Balise(5, ['noir'])
balise6 = Balise(6, ['vert', 'rouge', 'noir'])
balise7 = Balise(7, ['vert'])
balise8 = Balise(8, ['rouge'])
balise9 = Balise(9, ['rouge'])
balise10 = Balise(10, ['vert', 'noir'])
balise11 = Balise(11, ['rouge'])
balise12 = ...

balise1.voisines = [balise2, balise3]
balise2.voisines = [balise1, balise4]
balise3.voisines = [balise1, balise6]
balise4.voisines = [balise2, balise5, balise6]
balise5.voisines = [balise4, balise10]
balise6.voisines = [balise3, balise4, balise7, balise11]
balise7.voisines = [balise6, balise10]
balise8.voisines = [balise9]
...
balise10.voisines = [balise5, balise7, balise12]
balise11.voisines = [balise6, balise9]
balise12.voisines = [balise9, balise10]
```