Exercices sur les graphes

Exercice 1 : *Cet exercice porte sur les graphes, la programmation, la structure de pile et l'algorithmique des graphes.*
On s'intéresse à la fabrication de pain. La recette est fournie sous la forme de tâches à réaliser. Cette recette est réalisée par une personne seule.

a) Préparer 500g de farine.
b) Préparer 1/3 de litre d'eau (33cl).
c) Préparer 1 c. à café de sel.
d) Préparer 20g de levure de boulanger.
e) Faire tiédir l'eau dans une casserole.
f) Délayer la levure dans l'eau tiède.
g) Laisser reposer la levure 5 minutes.
h) Préparer un grand saladier.
i) Verser la farine dans le saladier.
j) Verser le sel dans le saladier.
k) Mélanger la farine et le sel puis creuser un puits.
l) Verser l'eau mélangée à la levure dans le puits.
m) Pétrir jusqu'à obtenir une pâte homogène.
n) Couvrir à l'aide d'un linge humide et laisser fermenter au moins 1h30.
o) Disposer dans le fond du four un petit récipient contenant de l'eau.
p) Préchauffer un four à 200 degrés Celsius.
q) Fariner un plan de travail.
r) Verser la pâte à pain sur le plan de travail.
s) Pétrir rapidement la pâte à pain.
t) Disposer la pâte dans un moule à cake.
u) Mettre au four pour 15 à 20 minutes, arrêter le four et sortir le pain.

La figure 1 représente les différentes tâches et les dépendances entre ces tâches sous la forme d'un graphe. Chaque sommet du graphe représente une tâche à réaliser. Les dépendances entre les tâches sont représentées par les arcs entre les sommets.
Par exemple, il y a une flèche sur l'arc qui part du sommet d'étiquette (l) et qui atteint le sommet d'étiquette (m) car il faut avoir réalisé la tâche *"Verser l'eau mélangée à la levure dans le puits."* (l) avant de pouvoir réaliser la tâche *"Pétrir jusqu'à obtenir une pâte homogène."* (m).

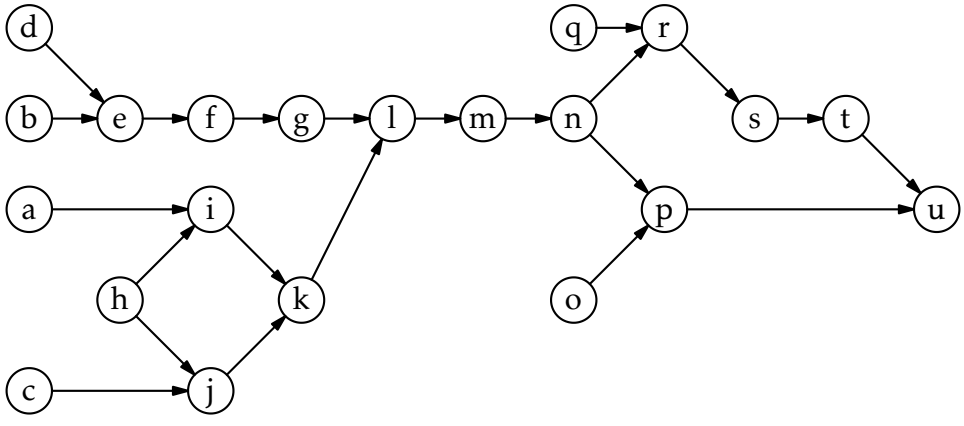


Figure 1. Recette du pain : tâches à effectuer avec leurs dépendances

1) Dire, sans justifier, s'il s'agit d'un graphe orienté ou non orienté.
   Solution : Il est orienté puisqu'il y a des flèches sur les arêtes, qui sont donc des arcs.
2) D'après le graphe, dire s'il est possible d'effectuer les réalisations dans chacun des ordres suivants :
   - réaliser la tâche (f) puis la tâche (g) **Solution :** oui
   - réaliser la tâche (g) puis la tâche (f) **Solution :** non
   - réaliser la tâche (i) puis la tâche (j) **Solution :** oui
   - réaliser la tâche (j) puis la tâche (i) **Solution :** oui
3) Donner toutes les tâches qu'il faut nécessairement avoir réalisées depuis le début pour pouvoir réaliser la tâche (k). Ne donner que les tâches nécessaires.
   Solution : Il faut réaliser a, c, h, i, j.
4) Indiquer, sans justifier, si le graphe de la figure 1 contient un cycle.
   Solution : Il n'y a pas de cycles.

Graphe des tâches

On s'intéresse désormais de manière plus générale à un graphe de tâches avec des dépendances.

Les sommets sont nommés par des indices. Comme précédemment, un arc orienté d'un sommet d'indice `i` à un sommet d'indice `j` signifie que la tâche représentée par le sommet d'indice `i` doit être réalisée avant la tâche représentée par le sommet d'indice `j`.

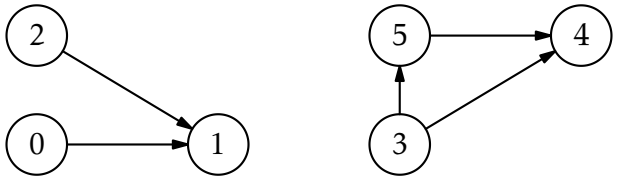
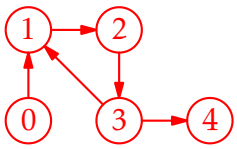


Figure 2. Exemple de graphe de dépendances entre 6 tâches

5) Déterminer un ordre permettant de réaliser toutes les tâches représentées dans le graphe de la figure 2 en respectant les dépendances entre les tâches.
   Solution : On peut faire 0, 2, 1, 3, 5, 4.

Voici une matrice d'adjacence d'un graphe écrite en langage Python et telle que si `M[i][j] = 1` alors il existe un arc qui va du sommet d'indice `i` au sommet d'indice `j`. Par exemple, `M[0][1] = 1` alors il existe un arc qui va du sommet d'indice `0` au sommet d'indice `1`.

```
M = [ [0, 1, 0, 0, 0],
      [0, 0, 1, 0, 0],
      [0, 0, 0, 1, 0],
      [0, 1, 0, 0, 1],
      [0, 0, 0, 0, 0] ]
```



6) Représenter le graphe associé à cette matrice d'adjacence. Les noms des sommets seront leurs indices.
7) Déterminer s'il est possible de trouver un ordre permettant de réaliser les tâches représentées par le graphe de la question 6 en respectant leurs dépendances. Si oui, donner l'ordre. Si non, expliquer pourquoi.
   Solution : Il y a le cycle $1 \to 2 \to 3 \to 1$ qui rend la réalisation impossible.

Voici le code Python d'une fonction `mystere`.

```python
def mystere(graphe, s, n, ouverts, fermes, resultat):
    """ Paramètres :
    graphe   un graphe représenté par une matrice d'adjacence
    s        l'indice d'un sommet du graphe
    n        le nombre de sommets du graphe
    ouverts  une liste de booléens permettant de savoir
             si le traitement d'un sommet a été commencé
    fermes   une liste de booléens permettant de savoir
             si le traitement d'un sommet a été terminé
    Retour : False s'il y a eu un "problème", True sinon.
    Le paramètre resultat sera modifié ultérieurement.
    """
    if ouverts[s]:
        return False
    if not fermes[s]:
        ouverts[s] = True
        for i in range(n):
            if graphe[s][i] == 1:
                val = mystere(graphe, i, n, ouverts, fermes, resultat)
                if not val:
                    return False
        ouverts[s] = False
        fermes[s] = True
        # ...
    return True
```

8) En utilisant la matrice `M` donnée précédemment, déterminer si la variable `ok` vaut **`True`** ou **`False`** à l'issue des instructions suivantes :

```python
n = len(M)
ouverts = [ False for i in range(n) ]
fermes = [ False for i in range(n) ]
ok = mystere(M, 1, n, ouverts, fermes, None)
```

Décrire précisément les appels effectués à la fonction `mystere` et les valeurs des tableaux `ouverts` et `fermes` lors de chaque appel. On pourra recopier et compléter le tableau ci-dessous.

Appel `mystere`	variable `ouverts`	variable `fermes`
Avant l'appel `mystere`	`[F,F,F,F,F]`	`[F,F,F,F,F]`
`mystere(M,1,5,[F,F,F,F,F],[F,F,F,F,F],None)`	`[F,T,F,F,F]`	`[F,F,F,F,F]`
`mystere(M,2,5,[F,T,F,F,F],[F,F,F,F,F],None)`	`[F,T,T,F,F]`	`[F,F,F,F,F]`
`mystere(M,3,5,[F,T,T,F,F],[F,F,F,F,F],None)`	`[F,T,T,T,F]`	`[F,F,F,F,F]`
`mystere(M,1,5,[F,T,T,T,F],[F,F,F,F,F],None)`	`[F,T,T,T,F]`	`[F,F,F,F,F]`

Lors du dernier appel, on revient sur un sommet qui est "ouvert". La fonction renvoie donc **`False`**.

9) De manière générale, expliquer dans quel cas cette fonction mystere renvoie **`False`**.
Solution : Elle renvoie **`False`** s'il y a un cycle puisqu'on revient sur un sommet qui est dans le chemin qu'on est en train d'explorer.

L'objectif est d'utiliser la fonction `mystere` pour écrire une fonction `ordre_realisation` qui, lorsque c'est possible, détermine l'ordre de réalisation des tâches d'un graphe donné par sa matrice d'adjacence en respectant les dépendances entre les tâches.
Une structure de données de pile est représentée par une classe `Pile` qui possède les méthodes suivantes :

- la méthode `estVide` qui renvoie `True` si la pile représentée par l'objet est vide, `False` sinon ;
- la méthode `empiler` qui prend en paramètre un élément et l'ajoute au sommet de la pile ;
- la méthode `depiler` qui renvoie la valeur du sommet de la pile et enlève cet élément.

10) Déterminer la valeur associée à la variable `elt` après l'exécution des instructions suivantes :

```
>>> essai = Pile()
>>> essai.empiler(3)
>>> essai.empiler(2)
>>> essai.empiler(10)
>>> elt = essai.depiler()
>>> elt = essai.depiler()
```

Solution : Elle contient 2.

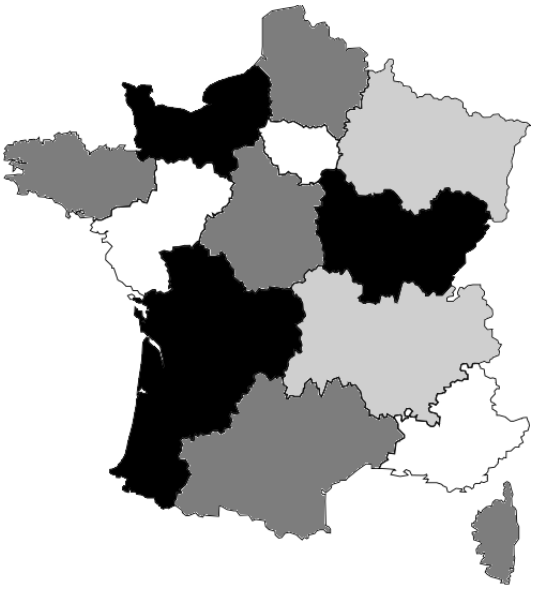
Lorsqu'il en existe un, un ordre de réalisation des tâches sera représenté par un objet de classe `Pile` contenant tous les sommets du graphe de manière à ce que les tâches qu'il faut réaliser en premier se retrouvent au sommet de la pile.
La fonction `ordre_realisation` est écrite de la manière suivante :

```
1  def ordre_realisation(graphe):
2      n = len(graphe)
3      ouverts = [ False for i in range(n) ]
4      fermes = [ False for i in range(n) ]
5      ordre = Pile()
6      ok = True
7      s = 0
8      while (ok and s < n):
9          ok = mystere(graphe, s, n, ouverts, fermes, ordre)
10         s = s + 1
11     if ok:
12         return ordre
13     return None
```

11) Sachant que dans la fonction mystere, la ligne 24 peut être remplacée par une ou plusieurs instructions, donner ce qu'il faut écrire pour que, lorsque c'est possible, `ordre_realisation` renvoie effectivement un ordre de réalisation des tâches du graphe.
Solution : Il suffit de rajouter `resultat.empiler(s)`. On rajoute la tâche actuelle au dessus de toutes les tâches suivantes.

Exercice 2 : *Cet exercice traite de programmation orientée objet en Python et d'algorithmique.*

Un pays est composé de différentes régions. Deux régions sont voisines si elles ont au moins une frontière en commun. L'objectif est d'attribuer une couleur à chaque région sur la carte du pays sans que deux régions voisines aient la même couleur et en utilisant le moins de couleurs possibles.
La figure 1 ci-dessous donne un exemple de résultat de coloration des régions de la France métropolitaine.



On rappelle quelques fonctions et méthodes des tableaux (le type **list** en Python) qui pourront être utilisées dans cet exercice :

- **len**(tab) : renvoie le nombre d'éléments du tableau `tab` ;
- `tab.append(elt)` : ajoute l'élément `elt` en fin de tableau `tab` ;
- `tab.remove(elt)` : enlève la première occurrence de `elt` de `tab` si `elt` est dans `tab`. Provoque une erreur sinon.

Exemples :

- **len**(`[1, 3, 12, 24, 3]`) renvoie `5` ;
- avec `tab=[1, 3, 12, 24, 3]`, l'instruction `tab.append(7)` modifie `tab` en `[1, 3, 12, 24, 3, 7]` ;
- avec `tab=[1, 3, 12, 24, 3]`, l'instruction `tab.remove(3)` modifie `tab` en `[1, 12, 24, 3]`.

Les deux parties de cet exercice forment un ensemble. Cependant, il n'est pas nécessaire d'avoir répondu à une question pour aborder la suivante. En particulier, on pourra utiliser les méthodes des questions précédentes même quand elles n'ont pas été codées.
Pour chaque question, toute trace de réflexion sera prise en compte.

Partie 1

On considère la classe `Region` qui modélise une région sur une carte et dont le début de l'implémentation est :

```
1  class Region:
2      '''Modélise une région d'un pays sur une carte.'''
3      def __init__(self, nom_region):
4          '''
5          initialise une région
6          : param nom_region (str) le nom de la région
7          '''
8          self.nom = nom_region
9          # tableau des régions voisines, vide au départ
10         self.tab_voisines = []
11         # tableau des couleurs disponibles pour colorier la région
12         self.tab_couleurs_disponibles = ['rouge', 'vert', 'bleu',
13                                          'jaune', 'orange', 'marron']
14         # couleur attribuée à la région et non encore choisie au départ
15         self.couleur_attribuee = None
```

1) Associer, en vous appuyant sur l'extrait de code précédent, les noms `nom`, `tab_voisines`, `tab_couleurs_disponibles` et `couleur_attribuee` au terme qui leur correspond parmi : *objet*, *attribut*, *méthode* ou *classe*. Ce sont des attributs.
2) Indiquer le type du paramètre `nom_region` de la méthode `__init__` de la classe `Region`. C'est un texte de type `str`.
3) Donner une instruction permettant de créer une instance nommée `ge` de la classe `Region` correspondant à la région dont le nom est "Grand Est".
```
ge = Region("Grand Est")
```
4) Recopier et compléter la ligne 7 de la méthode de la classe `Region` ci-dessous :

```
1    def renvoie_premiere_couleur_disponible(self):
2        '''
3        Renvoie la première couleur du tableau des couleurs
4        disponibles supposé non vide.
5        : return (str)
6        '''
7        return ...
```

```
    def renvoie_premiere_couleur_disponible(self):
        return self.tab_couleurs_disponibles[0]
```

5) Recopier et compléter la ligne 6 de la méthode de la classe `Region` ci-dessous :

```
1    def renvoie_nb_voisines(self) :
2        '''
3        Renvoie le nombre de régions voisines.
4        : return (int)
5        '''
6        return ...
```

```
    def renvoie_nb_voisines(self):
        return len(self.tab_voisines)
```

6) Compléter la méthode de la classe `Region` ci-dessous à partir de la ligne 7 :

```
1    def est_coloriee(self):
2        '''
3        Renvoie True si une couleur a été attribuée à cette
4        région et False sinon.
5        : return (bool)
6        '''
7        ...
```

```
    def est_coloriee(self):
        return self.couleur_attribuee is not None
```

7) Compléter la méthode de la classe `Region` ci-dessous à partir de la ligne 9 :

```
1    def retire_couleur(self, couleur):
2        '''
3        Retire couleur du tableau de couleurs disponibles de
4        la région si elle est dans ce tableau. Ne fait rien sinon.
5        : param couleur (str)
6        : ne renvoie rien
7        : effet de bord sur le tableau des couleurs disponibles
```

```
8         '''
9         ...
```

```
    def retire_couleur(self, couleur):
        if couleur in self.tab_couleurs_disponibles:
            self.tab_couleurs_disponibles.remove(couleur)
```

8) Compléter la méthode de la classe `Region` ci-dessous, à partir de la ligne 8, en utilisant une boucle :

```
1     def est_voisine(self, region):
2         '''
3         Renvoie True si la region passée en paramètre est une
4         voisine et False sinon.
5         : param region (Region)
6         : return (bool)
7         '''
8         ...
```

```
    def est_voisine(self, region):
        for v in self.tab_voisines:
            if v == region:
                return True
        return False
```

Partie 2

Dans cette partie :

- on considère qu'on dispose d'un ensemble d'instances de la classe `Region` pour lesquelles l'attribut `tab_voisines` a été renseigné ;
- on pourra utiliser les méthodes de la classe `Region` évoquées dans les questions de la partie 1 :
  - `renvoie_premiere_couleur_disponible`
  - `renvoie_nb_voisines`
  - `est_coloriee`
  - `retire_coule`

On a créé une classe `Pays` :

- cette classe modélise la carte d'un pays composé de régions ;
- l'unique attribut `tab_regions` de cette classe est un tableau (type `list` en Python) dont les éléments sont des instances de la classe `Region`.

9) Recopier et compléter la méthode de la classe `Pays` ci-dessous à partir de la ligne 7 :

```
1     def renvoie_tab_regions_non_coloriees(self):
2         '''
3         Renvoie un tableau dont les éléments sont les régions
4         du pays sans couleur attribuée.
5         : return (list) tableau d'instances de la classe Region
6         '''
7         ...
```

```
    def renvoie_tab_regions_non_coloriees(self):
        resultats = []
        for region in self.tab_regions:
            if not region.est_coloriee:
                resultats.append(region)
        return resultats
```

10) On considère la méthode de la classe `Pays` ci-dessous.

```
1     def renvoie_max(self):
2         nb_voisines_max = -1
3         region_max = None
4         for reg in self.renvoie_tab_regions_non_coloriees():
5             if reg.renvoie_nb_voisines() > nb_voisines_max:
6                 nb_voisines_max = reg.renvoie_nb_voisines()
7                 region_max = reg
8         return region_max
```

a) Expliquer dans quel cas cette méthode renvoie **None**.
La méthode renvoie **None** lorsque toutes les régions ont été coloriées.

b) Indiquer, dans le cas où cette méthode ne renvoie pas **None**, les deux particularités de la région renvoyée.
La région renvoyée n'est pas coloriée et a le nombre de voisines maximal.

11) Coder la méthode `colorie(self)` de la classe `Pays` qui choisit une couleur pour chaque région du pays de la façon suivante :

- On récupère la région non coloriée qui possède le plus de voisines.
- Tant que cette région existe :
  - La couleur attribuée à cette région est la première couleur disponible dans son tableau de couleurs disponibles.
  - Pour chaque région voisine de la région :
    - si la couleur choisie est présente dans le tableau des couleurs disponibles de la région voisine alors on la retire de ce tableau.
  - On récupère à nouveau la région non coloriée qui possède le plus de voisines.

```
    def colorie(self):
        region = self.renvoie_max()
        while region is not None:
            couleur = region.renvoie_premiere_couleur_disponible()
            region.couleur_attribuee = couleur
            for voisine in region.tab_voisines:
                voisine.retire_couleur(couleur)
            region = self.renvoie_max()
```

Exercice 3 : *Cet exercice porte sur la programmation orientée objet, les graphes et utilise la structure de données dictionnaire.*

La direction de la station de ski *Le Lièvre Blanc*, spécialisée dans la pratique du ski de fond, souhaite disposer d'un logiciel lui permettant de gérer au mieux son domaine skiable. Elle confie à un développeur informatique la mission de concevoir ce logiciel. Celui-ci décide de caractériser les pistes de ski à l'aide d'une classe `Piste` et le domaine de ski par une classe `Domaine`.

Le code Python de ces deux classes est donné en Annexe.

Partie A – Analyse des classes Piste et Domaine

1) Lister les attributs de la classe Piste en précisant leur type.

Solution : Les attributs de la classe `Piste` sont :

Attribut	Type
`nom`	`str`
`denivele`	`int`
`longueur`	`float`
`couleur`	`str`
`ouverte`	`bool`

La difficulté des pistes de ski de fond est représentée par 4 couleurs : verte, bleue, rouge et noire. La piste verte est considérée comme très facile, la piste bleue comme facile, la piste rouge de difficulté moyenne et la piste noire difficile. Dans la station de ski *Le Lièvre blanc*, l'équipe de direction décide de s'appuyer uniquement sur le dénivelé pour attribuer la couleur d'une piste de ski.
Ainsi, une piste de ski sera de couleur :

- `'noire'` si son dénivelé est supérieur ou égal à 100 mètres ;
- `'rouge'` si son dénivelé est strictement inférieur à 100 mètres, mais supérieur ou égal à 70 mètres ;
- `'bleue'` si son dénivelé est strictement inférieur à 70 mètres, mais supérieur ou égal à 40 mètres ;
- `'verte'` si son dénivelé est strictement inférieur à 40 mètres.

2) Écrire la méthode `set_couleur` de la classe `Piste` qui permet d'affecter à l'attribut `couleur` la chaîne de caractères correspondant à la couleur de la piste.

Solution :

```
class Piste:
    def __init__(self, nom, denivele, longueur):
        ...

    def set_couleur(self):
        if self.denivele >= 100:
            self.couleur = 'noire'
        elif self.denivele >= 70:
            self.couleur = 'rouge'
        elif self.denivele >= 40:
            self.couleur = 'bleue'
        else:
            self.couleur = 'verte'
```

On exécute à présent le programme suivant afin d'attribuer la couleur adéquate à chacune des pistes du domaine skiable *Le Lièvre Blanc*.

```
for piste in lievre_blanc.get_pistes():
    piste.set_couleur()
```

3) Indiquer, parmi les 4 propositions ci-dessous, le type de l'élément renvoyé par l'instruction Python `lievre_blanc.get_pistes()`.

- Proposition A : une chaîne de caractères ;
- Proposition B : un objet de type Piste ;
- Proposition C : une liste de chaînes de caractères ;

- Proposition D : une liste d'objets de type Piste.

En raison d'un manque d'enneigement, la direction de la station est souvent contrainte de fermer toutes les pistes vertes car elles sont situées généralement en bas du domaine.

4) Écrire un programme Python dont l'exécution permet de procéder à la fermeture de toutes les pistes vertes en affectant la valeur **False** à l'attribut `ouverte` des pistes concernées.

Solution : On peut écrire :

```
for piste in lievre_blanc.get_pistes():
    if piste.couleur == 'verte':
        piste.ouverte = False
```

5) Écrire une fonction `pistes_de_couleur` prenant pour paramètres une chaîne de caractères `couleur` représentant la difficulté d'une piste et une liste `lst` de pistes de ski de fond. Cette fonction renvoie la liste des noms des pistes dont `couleur` est le niveau de difficulté.

Exemple : l'instruction `pistes_de_couleur(lievre_blanc.get_pistes(), 'noire')` renvoie la liste `['Petit Bonheur', 'Forêt', 'Duvallon']`.

Solution : On peut écrire :

```
def pistes_de_couleur(couleur, lst):
    res = []
    for piste in lst:
        if piste.couleur == couleur:
            res.append(piste.nom)
    return res
```

Un skieur de bon niveau se prépare assidûment pour le prochain semi-marathon, d'une distance de 21,1 kilomètres. À chaque entraînement, il note la liste des noms des pistes qu'il a parcourues et il souhaite disposer d'un outil lui indiquant si la distance totale parcourue est au moins égale à la distance qu'il devra parcourir le jour du semi-marathon.

La fonction `semi_marathon` donnée ci-dessous répond aux attentes du skieur : cette fonction prend en paramètre une liste `L` de noms de pistes et renvoie un booléen égal à **True** si la distance totale parcourue est strictement supérieure à 21,1 kilomètres, **False** sinon.

```
def semi_marathon(L):
    distance = 0
    liste_pistes = lievre_blanc.get_pistes()
    for nom in L:
        for piste in liste_pistes:
            if piste.get_nom() == nom:
                distance = distance + piste.get_longueur()
return ...
```

On donne ci-dessous deux exemples d'appels à cette fonction :

```
>>> entrainement1 = ['Verneys', 'Chateau enneigé', 'Rois mages', 'Diablotin']
>>> semi_marathon(entrainement1)
True
>>> entrainement2 = ['Esseillon', 'Aigle Royal', 'Duvallon']
>>> semi_marathon(entrainement2)
False
```

6) Recopier et compléter la fonction `semi_marathon`.

Partie B – Recherche par force brute

Le plan des pistes du domaine Le Lièvre Blanc peut être représenté par le graphe suivant :

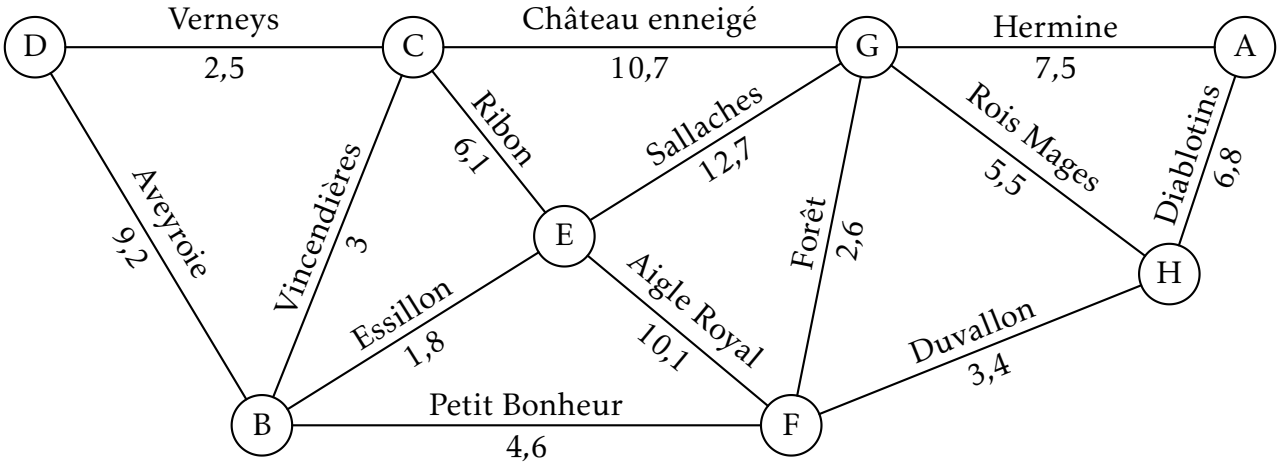


Figure 1. Graphe du domaine *Le Lièvre Blanc*

Sur chaque arête, on a indiqué le nom de la piste et sa longueur en kilomètres. Les sommets correspondent à des postes de secours.

Un pisteur-secouriste de permanence au point de secours D est appelé pour une intervention en urgence au point de secours A. La motoneige de la station étant en panne, il ne peut s'y rendre qu'en skis de fond. Il décide de minimiser la distance parcourue et cherche à savoir quel est le meilleur parcours possible. Pour l'aider à répondre à ce problème, on décide d'implémenter le graphe ci-dessus grâce au dictionnaire de dictionnaires suivant :

```
domaine = {'A' : {'G': 7.5, 'H': 6.8},
           'B' : {'C': 3.0, 'D': 9.2, 'E': 1.8, 'F': 4.6},
           'C' : {'B': 3.0, 'D': 2.5, 'E': 6.1, 'G': 10.7},
           'D' : {'B': 9.2, 'C': 2.5},
           'E' : {'B': 1.8, 'C': 6.1, 'F': 10.1, 'G': 12.7},
           'F' : {'B': 4.6, 'E': 10.1, 'G': 2.6, 'H': 3.4},
           'G' : {'A': 7.5, 'C': 10.7, 'E': 12.7, 'F': 2.6, 'H': 5.5},
           'H' : {'A': 6.8, 'F': 3.4, 'G': 5.5} }
```

7) Écrire une instruction Python permettant d'afficher la longueur de la piste allant du sommet `'E'` au sommet `'F'`.

   Solution : `domaine['E']['F']`

8) Écrire une fonction `voisins` qui prend en paramètres un graphe `G` et un sommet `s` du graphe `G` et qui renvoie la liste des voisins du sommet `s`.

   Exemple : l'instruction `voisins(domaine, 'B')` renvoie la liste `['C', 'D', 'E', 'F']`.

   Solution : On peut écrire :

```
def voisins(G, sommet):
    return [s for s in G[sommet]]

# ou

def voisins(G, sommet):
    vois = []
    for s in G[sommet]:
        vois.append(s)
    return vois
```

9) Recopier et compléter la fonction `longueur_chemin` donnée ci-dessous : cette fonction prend en paramètres un graphe `G` et un chemin du graphe `G` sous la forme d'une liste de sommets et renvoie sa longueur en kilomètres.
Exemple : l'instruction `longueur_chemin(domaine, ['B', 'E', 'F', 'H'])` renvoie le nombre flottant `15.3`.

```
def longueur_chemin(G, chemin):
    precedent = chemin[0]
    longueur = 0
    for i in range(1, len(chemin)):
        longueur = longueur + G[precedent][chemin[i]]
        precedent = chemin[i]
    return longueur
```

On donne ci-dessous une fonction `parcours` qui renvoie la liste de tous les chemins du graphe `G` partant du sommet `depart` et parcourant les sommets de façon unique, c'est-à-dire qu'un sommet est atteint au plus une fois dans un chemin.
Par exemple, l'appel `parcours(domaine, 'A')` renvoie la liste de tous les chemins partant du sommet A dans le graphe `domaine` sans se soucier ni de la longueur du chemin, ni du sommet d'arrivée. Ainsi, `['A', 'G', 'C']` est un chemin possible, tout comme `['A', 'G', 'C', 'B', 'E', 'F', 'H']`.

```
def parcours(G, depart, chemin=[], lst_chemins=[]):
    if chemin == []:
        chemin = [depart]
    for sommet in voisins(G, depart):
        if sommet not in chemin:
            lst_chemins.append(chemin + [sommet])
            parcours(G, sommet, chemin + [sommet])
    return lst_chemins
```

10) Expliquer en quoi la fonction `parcours` est une fonction récursive.
Solution : La fonction `parcours` s'appelle elle-même à la ligne 7. Elle est donc récursive.

Un appel à la fonction `parcours` précédente renvoie une liste de chemins dans laquelle figurent des doublons.

11) Recopier et compléter la fonction `parcours_dep_arr` ci-après qui renvoie la liste des chemins partant du sommet depart et se terminant par le sommet arrivee dans le graphe G entrés en paramètres. La liste renvoyée ne doit pas comporter de doublons. Attention, plusieurs lignes de code sont nécessaires.

```
def parcours_dep_arr(G, depart, arrivee):
    liste = parcours(G, depart)
    chemins = []
    for c in liste:
        if c[-1] == arrivee and c not in chemins:
            chemins.append(c)
    return chemins
```

12) Recopier et compléter la fonction `plus_court` donnée ci-dessous. La fonction `plus_court` prend pour paramètres un graphe `G`, un sommet de départ `depart` et un sommet d'arrivée `arrivee` ; elle renvoie un des chemins les plus courts sous la forme d'une liste de sommets.
Remarque : En fait, la fonction `parcours` ne renvoie pas de doublons, mais on va faire comme si c'était le cas.

```
def plus_court(G, depart, arrivee):
    liste_chemins = parcours_dep_arr(G, depart, arrivee)
    chemin_plus_court = liste_chemins[0]
    minimum = longueur_chemin(G, chemin_plus_court)
    for chemin in liste_chemins:
        longueur = longueur_chemin(G, chemin)
        if longueur < minimum:
            minimum = longueur
            chemin_plus_court = chemin
    return chemin_plus_court
```

13) Expliquer en quoi le choix fait par le pisteur-secouriste de choisir la distance minimale pour arriver le plus rapidement possible sur le lieu de l'incident est discutable. Proposer un meilleur critère de choix.

Solution : Le temps de trajet n'est pas forcément proportionnel à la distance. Il faudrait tenir compte du dénivelé ou directement du temps mis pour parcourir chaque liaison entre 2 sommets.

Annexe

```
# Pistes
class Piste:
    def __init__(self, nom, denivele, longueur):
        self.nom = nom
        self.denivele = denivele  # en mètres
        self.longueur = longueur  # en kilomètres
        self.couleur = ''
        self.ouverte = True

    def get_nom(self):
        return self.nom

    def get_longueur(self):
        return self.longueur

    def set_couleur(self):
        # À compléter

    def get_couleur(self):
        return self.couleur

# Domaine skiable
class Domaine:
    def __init__(self, a):
        self.nom = a
        self.pistes = []

    def ajouter_piste(self, nom_piste, denivele, longueur):
        self.pistes.append(Piste(nom_piste, denivele, longueur))

    def get_pistes(self):
        return self.pistes

# Programme principal
lievre_blanc = Domaine("Le Lièvre Blanc")
lievre_blanc.ajouter_piste('Aveyrole', 62, 9.2)
lievre_blanc.ajouter_piste('Verneys', 10, 2.5)
lievre_blanc.ajouter_piste('Vincendières', 45, 3)
lievre_blanc.ajouter_piste('Ribon', 70, 6.1)
lievre_blanc.ajouter_piste('Esseillon', 8, 1.8)
lievre_blanc.ajouter_piste('Petit Bonheur', 310, 4.6)
lievre_blanc.ajouter_piste('Aigle Royal', 85, 10.1)
lievre_blanc.ajouter_piste('Château enneigé', 54, 10.7)
lievre_blanc.ajouter_piste('Sallanches', 78, 12.7)
lievre_blanc.ajouter_piste('Forêt', 145, 2.6)
lievre_blanc.ajouter_piste('Hermine', 27, 7.5)
lievre_blanc.ajouter_piste('Rois mages', 42, 5.5)
lievre_blanc.ajouter_piste('Diablotin', 76, 6.8)
lievre_blanc.ajouter_piste('Duvallon', 200, 3.4)
```