ÉPREUVE MUTUALISÉE AVEC E3A-POLYTECH

ÉPREUVE SPÉCIFIQUE - FILIÈRE PC

INFORMATIQUE

Durée : 3 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

RAPPEL DES CONSIGNES

- *Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, bleu clair ou turquoise, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.*
- *Ne pas utiliser de correcteur.*
- *Écrire le mot FIN à la fin de votre composition.*

Les calculatrices sont interdites.

Le sujet est composé de trois parties.

L'épreuve est à traiter en langage **Python** sauf pour les bases de données.
Les différents algorithmes doivent être rendus dans leur forme définitive sur le **Document Réponse** dans l'espace réservé à cet effet en respectant les éléments de syntaxe du langage (les brouillons ne sont pas acceptés).
La réponse ne doit pas se limiter à la rédaction de l'algorithme sans explication, les programmes doivent être expliqués et commentés de manière raisonnable.

Énoncé et Annexe : 16 pages
Document Réponse (DR) : 11 pages

Seul le Document Réponse (DR) doit être rendu dans son intégralité (le QR Code doit être collé sur la première page du DR.

Gestion de Randonnées

Les fonctions seront définies avec leur signature dans le sujet :

`ma_fonction(arg1:type1, arg2:type2) -> type3.`

Cette notation permet de définir une fonction qui se nomme `ma_fonction` qui prend deux arguments en entrée, `arg1` de type `type1` et `arg2` de type `type2`. Cette fonction renvoie une valeur de type `type3`.

Il ne faut pas recopier les signatures des fonctions dans le **DR**, il faut écrire directement :

```
def ma_fonction(arg1, arg2):
    # liste d'instructions
```

Introduction

Lors d'une randonnée, des applications disponibles sur smartphones permettent d'enregistrer l'itinéraire parcouru. L'utilisation de ces données peut permettre d'analyser *a posteriori* le parcours effectué (distance, durée, dénivelés…) ou de planifier de nouvelles sorties.

L'objectif du travail proposé est de découvrir différentes facettes de ces applications.

Le sujet abordera les points suivants :

- l'organisation des différents parcours dans une base de données,
- différentes stratégies pour obtenir des informations sur le dénivelé effectué,
- la planification de randonnées en utilisant des algorithmes de type Dijkstra.

Les données recueillies lors d'une randonnée sont généralement stockées dans un fichier au format GPX (pour GPS eXchange Format). Ce fichier est appelé *trace* GPX de la randonnée.

Partie I - Gestion des randonnées dans une base de données

Les applications de randonnées permettent à un utilisateur de stocker les données de ses randonnées effectuées ou d'avoir accès à celles effectuées par d'autres utilisateurs. Ces données sont stockées dans une base contenant notamment les tables suivantes.

La table `Randonnee` contenant :

`Id`	entier, identifiant de la randonnée ;
`Titre`	chaîne de caractères, titre de la randonnée ;
`Type`	chaîne de caractères du type de l'activité : "Pied", "VTT", "Cheval" ;
`Lieu`	chaîne de caractères, coordonnées GPS du point de départ ;
`Distance`	flottant, longueur en kilomètres de la randonnée ;
`DenP`	entier, dénivelé positif en mètres ;
`DenN`	entier, dénivelé négatif en mètres ;
`Duree`	entier, durée en minutes de la randonnée ;
`Niveau`	entier compris entre 1 et 5 (1 : facile à 5 : extrême), difficulté ;
`IdAuteur`	entier, identifiant de l'auteur de la randonnée ;
`Trace`	chaîne de caractères, lien internet vers la trace GPX.

La table `Auteur` contenant :

`Id`	entier, identifiant de l'auteur (le randonneur) ;
`Nom`	chaîne de caractères, nom de l'auteur ;
`Prenom`	chaîne de caractères, prénom de l'auteur ;
`Pseudo`	chaîne de caractères, pseudo de l'auteur ;
`Mail`	chaîne de caractères, mail de l'auteur.

Q1. Expliquer en quoi l'attribut `Titre` ne peut probablement pas être une clé primaire pour la table `Randonnee`. Proposer un attribut de la table `Randonnee` qui puisse être une clé primaire.

Q2. Identifier un attribut qui soit une clé étrangère de la table `Randonnee`.

Q3. Écrire une requête SQL dont l'évaluation renvoie le titre, les coordonnées GPS du point de départ et la longueur des randonnées à pied.

Q4. Écrire une requête SQL dont l'évaluation renvoie l'identifiant de l'auteur et son nombre d'activités à pied de niveau 3, classées par ordre décroissant du nombre d'activités de chaque auteur.

Q5. Écrire une requête SQL dont l'évaluation renvoie le `Pseudo` de l'auteur et le `Titre` des randonnées stockées dans la base.

Q6. Écrire une requête SQL dont l'évaluation renvoie les nom et prénom d'un des auteurs ayant posté le plus de randonnées à cheval.

Partie II - Quelques calculs de dénivelés

Nous allons maintenant nous intéresser plus particulièrement aux données enregistrées par l'application lors d'une randonnée. En pratique, une application enregistre régulièrement les données fournies par le GPS du téléphone portable comme la latitude et la longitude exprimées en degrés ainsi que l'altitude (élévation) exprimée en mètres. La **figure 1** présente un extrait du fichier *trace* GPX.

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx
  xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/11.xsd"
  xmlns="http://www.topografix.com/GPX/1/1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <trk>
    <name>Randonne</name>
    <type>Marche</type>
    <trkseg>
      <trkpt lat="43.331146650016307830810546875" lon="-1.627655038610100746154785156 25">
        <ele>261</ele>
        <time>2022-08-01T07:37:39.000Z</time>
      </trkpt>
      <trkpt lat="43.331055035814642906188964843 75" lon="-1.627895934507250785827636718 75">
        <ele>267.20001220703125</ele>
        <time>2022-08-01T07:37:52.000Z</time>
      </trkpt>
      <trkpt lat="43.331047911196947097778320312 5" lon="-1.627927953377366065979003906 25">
        <ele>267.79998779296875</ele>
        <time>2022-08-01T07:37:54.000Z</time>
      </trkpt>
      <trkpt lat="43.331040535122156143188476562 5" lon="-1.627966091036796569824218 75">
        <ele>268.399993896484375</ele>
        <time>2022-08-01T07:37:57.000Z</time>
      </trkpt>
```

Figure 1 - Exemple de fichier *trace* GPX

Le module `gpxpy` de Python permet de lire et extraire simplement les données de ce type de fichier.

Q7. Écrire une ligne de code permettant l'importation du module `gpxpy`.

Un *parcours* (ou une randonnée) est alors une succession de points. Après lecture et traitement du fichier à l'aide du module `gpxpy`, l'itinéraire d'une randonnée est représenté par une liste de points où chacun de ces points est un triplet de trois flottants correspondant respectivement à la latitude, la longitude et l'altitude. Le premier point d'un itinéraire sera le *point de départ* et le dernier le *point d'arrivée*.

Pour améliorer la lisibilité de la signature de nos fonctions, nous utiliserons le type `trpt` (pour track point ou point de la trace) pour représenter les triplets de points ainsi que le type `itineraire` pour les listes de points. Ainsi, à partir du fichier de la **figure 1** on pourra introduire les variables `p0`, `p1`, `p2`, `p3` qui sont de type `trpt` et la variable `iti` qui est de type `itineraire` :

```
p0 = (43.331146, -1.627655, 261.00) # point de depart
p1 = (43.331055, -1.627895, 267.20) # point intermediaire
p2 = (43.331047, -1.627927, 267.79) # point intermediaire
p3 = (43.331040, -1.627966, 268.39) # point d arrivee
iti = [p0, p1, p2, p3] # itineraire
```

Rappelons (**figure 2**) qu'un point de la surface terrestre est défini par :

- sa *latitude*, notée ϕ, qui est la mesure angulaire entre l'équateur et ce point. Elle est représentée par un angle compris entre $-90°$ et $+90°$;
- sa *longitude*, notée λ, qui est la mesure angulaire entre le méridien de référence (méridien de Greenwich) et ce point. Elle est représentée par un angle compris entre $-180°$ et $+180°$;
- son altitude qui exprime la hauteur entre le niveau de la mer et le niveau du point. Elle est représentée par un réel et s'exprime en mètres.

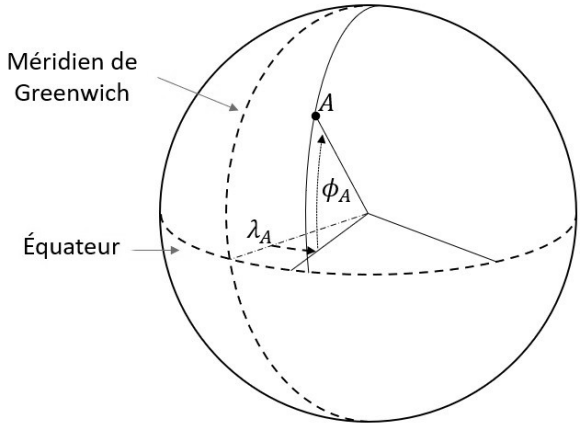

Figure 2 - Repérage, sur la surface du globe, d'un point A de latitude ϕ_A et de longitude λ_A

La syntaxe pour extraire les éléments d'un tuple est identique à celle utilisée pour les éléments d'une liste mais les tuples ne sont pas modifiables (ou mutables).

```
>>> p0 = (43.331146, -1.627655, 261.00) # point de depart
>>> p0[0]
43.331146
>>> (a, b, c) = p0
>>> a
43.331146
```

On considère la fonction `mystere`, dont l'argument `iti` est non vide :

```
1 def mystere(iti :itineraire) -> float :
2     s = 0
3     for i in range(len(iti)):
4         (lat, long, alt) = iti[i]
5         s = s + alt
6     return (s/len(iti))
```

Q8. Donner la valeur numérique que renvoie le code suivant. Donner la signification de cette valeur dans le contexte du sujet.

```
1 >>> p0 = (47.8741, 1.8758, 100)
2 >>> p1 = (47.8744, 1.8759, 102)
3 >>> p2 = (47.8748, 1.8761, 110)
4 >>> p3 = (47.8750, 1.8759, 108)
5 >>> l1 = [p0, p1, p2, p3]
6 >>> mystere(l1)
```

Q9. Donner la complexité temporelle de la fonction `mystere` en fonction de la taille n de la liste passée en argument.

Q10. Écrire une fonction `altitude_maximale(iti:itineraire) -> float` qui, étant donné une liste non vide `iti` de points, renvoie l'altitude maximale de l'itinéraire en mètres.

Le *dénivelé global* d'une randonnée est la différence entre l'altitude maximale et l'altitude du point de départ.

Q11. En utilisant la fonction `altitude_maximale`, écrire une fonction `denivele_global(iti:itineraire) -> float` qui, étant donné une liste `iti` de points, renvoie le dénivelé global de la randonnée.

II.1 - Premier calcul de dénivelé positif

Le dénivelé entre deux points successifs p_1 et p_2 d'un itinéraire est dit positif si la différence entre l'altitude de p_2 et l'altitude de p_1 est positive. On appelle alors *dénivelé positif* la différence entre ces deux altitudes. Le *dénivelé positif cumulé* d'une randonnée est la somme de tous les dénivelés positifs entre les points successifs du parcours.

Q12. Écrire une fonction `denivele_positif_cumule(iti:itineraire) -> float` qui, étant donné une liste `iti` de points, renvoie le dénivelé positif cumulé de la randonnée.

La méthode de mesure de l'altitude par le GPS est relativement imprécise due à la présence éventuelle d'une couverture nuageuse, d'un parcours sous des arbres... Or, lors du calcul du dénivelé positif cumulé, de faibles erreurs répétées peuvent induire une erreur conséquente sur le calcul cumulé.

Par exemple, si le randonneur effectue une randonnée en bord de mer sur une plage d'altitude constante mais que le GPS effectue des mesures erronées pour donner une liste d'altitudes égale à `[0, -2, 2, -2, 2, -2, 2, -2, 2]`, le dénivelé positif cumulé calculé par la fonction précédente est égal à 16 mètres alors qu'il devrait être nul.

Nous allons envisager deux méthodes pour pallier ces imprécisions : le lissage des altitudes et l'utilisation d'altitudes de référence.

II.2 - Lissage des altitudes

Le *lissage* d'une liste de longueur n des altitudes par moyenne glissante de pas p consiste à remplacer l'altitude au point numéroté i par la moyenne des altitudes des points numérotés $i, i+1, \ldots, i+j$ où $j = \min\{p-1, n-i-1\}$.

Par exemple, lorsque $p = 2$, la liste précédente sera remplacée par :

liste de départ	0	-2	2	-2	2	-2	2	-2	2
calcul	$\frac{0-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	2
liste lissée	-1	0	0	0	0	0	0	0	2

Le dénivelé positif est alors de 3 mètres, ce qui est plus proche de la réalité de l'itinéraire.

Q13. Écrire une fonction `alt_glissante(liste_alt:list, p:int) -> list` qui étant donné une liste d'altitudes et un entier `p`, crée une nouvelle liste contenant la moyenne glissante des altitudes avec un pas `p`. On pourra utiliser la fonction `min` qui prend deux nombres flottants en entrée et renvoie le plus petit des deux.

Q14. Évaluer la complexité de la fonction `alt_glissante` en fonction de la taille n de la liste d'altitudes passée en argument et du pas p.

II.3 - Utilisation d'altitudes de référence

Une autre stratégie pour améliorer la précision sur les altitudes, une fois la randonnée effectuée et une connexion internet plus stable trouvée, consiste à se connecter à une base de référence qui, étant donné un point du globe, renvoie son altitude. Bien sûr, tous les points ne sont pas stockés dans la base. Nous supposerons que la surface du globe est quadrillée par une liste de latitudes et une liste de longitudes et qu'en chaque point de cette grille l'altitude a été mesurée précisément. Ces altitudes sont stockées dans un dictionnaire nommé `dem` (Digital Elevation Model) dont les clés sont des couples latitude / longitude et les valeurs sont les altitudes correspondantes.

On considère le code suivant :

```
1 lat_ref = []
2 long_ref = []
3 for (lat, long) in dem:
4     lat_ref.append(lat)
5     long_ref.append(long)
```

On supposera dans la suite que les valeurs -90 et 90 sont dans `lat_ref` et que les valeurs -180 et 180 sont dans `long_ref`.

Q15. Indiquer le type des variables `lat_ref` et `long_ref`. Expliquer quel est le contenu de ces variables dans le contexte de ce sujet.

On considère le code suivant :

```
1  def auxiliaire(x, y):
2      if x == [] : return y
3      if y == [] : return x
4      if x[len(x)-1] < y[len(y)-1] :
5          val = y.pop()
6      else :
7          val = x.pop()
8      z = auxiliaire(x,y)
9      z.append(val)
10     return z
11
12 def principal(x):
13     if len(x) <= 1:
14         return x
15     else:
16         m = len(x)//2
17         x1 = principal(x[0:m])
18         y1 = principal(x[m:len(x)])
19         z = auxiliaire(x1, y1)
20         return z
```

Q16. Précisez la signature des fonctions `auxiliaire` et `principal`. La réponse doit être justifiée.

Q17. Sur le **DR**, cocher la (les) cases qui correspond(ent) au(x) type(s) de programmation utilisé(s) pour coder la fonction `principal`.

Q18. Justifier brièvement que, étant donné une liste `x`, l'appel `principal(x)` termine.

Q19. L'appel `principal(x)` permet de renvoyer une liste triée par ordre croissant. Proposer un nom qui décrit le type de tri utilisé en justifiant brièvement votre choix.

Par la suite, on suppose que les listes `lat_ref` et `long_ref` sont **triées**. Il faut maintenant déterminer le point du dictionnaire `dem` le plus proche d'un point donné.

On donne une implémentation partielle de la fonction `ref(valeur, liste_ref)` qui, étant donné un flottant `valeur` et une liste non vide de flottants triés par ordre croissant `liste_ref` tels que `valeur` est strictement compris entre le premier et le dernier élément de `liste_ref`, renvoie la valeur de la liste `liste_ref` la plus proche de `valeur`.

```
1  def ref(valeur:float, liste_ref:list) -> float:
2      # on détermine ind_deb et ind_fin tels que
3      # liste_ref[ind_deb]<valeur<=liste_ref[ind_fin]
4      # avec une méthode par dichotomie
5      ind_deb = ......
6      ind_fin = ......
7      while ind_deb < ind_fin - 1:
8          k = .....
9          if valeur <= liste_ref[k] :
10             ind_fin = .....
11         else:
12             .....
13     # on détermine le plus proche
14     if liste_ref[ind_fin]-valeur<valeur-liste_ref[ind_deb]:
15         return .....
16     else:
17         return .....
```

Q20. Compléter les lignes 5, 6, 8, 10, 12, 15 et 17 de la fonction `ref`.

Q21. Utiliser les données précédentes pour écrire une fonction `standardise(liste_parcours:itineraire) -> itineraire` qui, étant donné un itinéraire, renvoie un nouvel itinéraire où l'altitude de chaque point a été remplacée par l'altitude issue du dictionnaire `dem`.
Pour chaque point de l'itinéraire, on cherchera la latitude ϕ_r de référence la plus proche de sa latitude, la longitude λ_r de référence la plus proche de sa longitude et on remplacera son altitude par l'altitude du point de référence de coordonnées (ϕ_r, λ_r).
Les variables `lat_ref`, `long_ref` et `dem` sont définies globalement en dehors de la fonction et peuvent être utilisées directement.

Partie III - Organisation d'un Trek

Un randonneur souhaite effectuer un long Trek, c'est-à-dire une série de randonnées sur plusieurs jours. Il organise son parcours à partir d'un site qui propose différentes randonnées d'une journée. Chaque randonnée est définie par un niveau de difficulté allant de 1 (facile) à 5 (extrême). L'ensemble des données permet au randonneur d'établir un graphe où :

- les sommets représentent les points de départ / arrivée des randonnées,
- les arêtes représentent les randonnées possibles avec comme poids le niveau de la randonnée.

On suppose qu'il y a une unique randonnée qui relie 2 sommets du graphe.
On suppose que les randonnées peuvent être effectuées du point de départ vers le point d'arrivée ou du point d'arrivée vers le point de départ sans que la difficulté ne soit modifiée. Le graphe représentant les différentes randonnées sera donc non orienté.

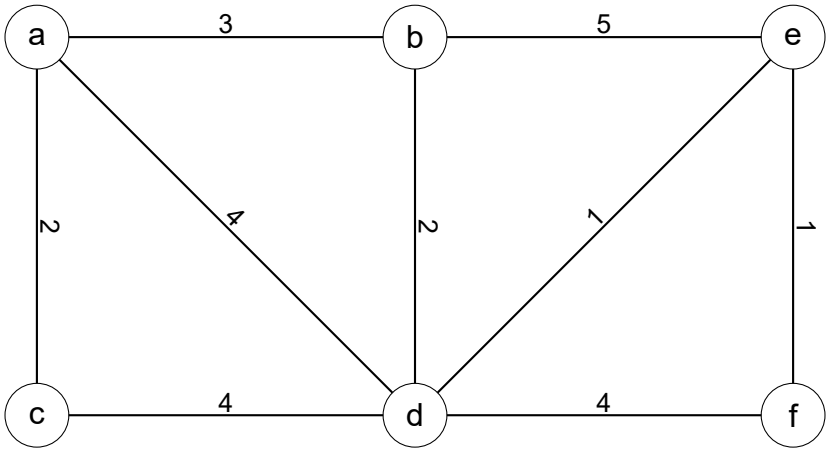

Figure 3 - Graphe G

Le graphe G de la **figure 3** est représenté par le dictionnaire `G` défini de la manière suivante :

```
G = dict()
G['a'] = {'b':3, 'c':2, 'd':4}
G['b'] = {'a':3, 'd':2, 'e':5}
G['c'] = {'a':2, 'd':4}
G['d'] = {'a':4, 'b':2, 'c':4, 'e':1, 'f':4}
G['e'] = {'b':5, 'd':1, 'f':1}
G['f'] = {'d':4, 'e':1}
```

Le randonneur souhaite aller du point `a` au point `f`. Cependant, comme il n'est pas entraîné, il choisit de trouver le chemin dont la somme des difficultés des randonnées est la plus petite, quel que soit le nombre d'étapes.

Afin d'utiliser le vocabulaire habituellement employé dans les algorithmes de parcours de graphes, on utilisera le terme "distance" au lieu du terme "difficulté" et on cherchera donc à minimiser la "distance" (au sens de "difficulté" du trek).

III.1 - Première idée : algorithme intuitif

Pour trouver son chemin, le randonneur exécute la fonction `mystere2` suivante.

```
1  def mystere2(graph:dict, Sd:str, Sf:str) -> tuple:
2      """graph : graphe représentant les randonnées disponibles
3      Sd : sommet de départ dans le graphe
4      Sf : sommet d'arrivée dans le graphe
5      Renvoie une liste de randonnées permettant de relier Sd à Sf ainsi
           que la somme des difficultés d'un tel chemin."""
6      dejaVisites = [] # Liste des sommets déjà visités
7      sTraite = Sd # Initialisation du sommet à traiter
8      chemin = [sTraite] # Chemin choisi initialisé
9      diffChemin = 0 # Difficulté du chemin choisi
10     # Construction iterative du chemin
11     while sTraite != Sf:
12         dejaVisites.append(sTraite)
13         d = float('inf') #valeur représentant l'infini
14         sInter = sTraite
15         for sommet in graph[sTraite]:
16             if sommet not in dejaVisites:
17                 if graph[sTraite][sommet] < d:
18                     sInter = sommet
19                     d = graph[sTraite][sommet]
20         diffChemin = diffChemin + d
21         chemin.append(sInter)
22         sTraite = sInter
23     return chemin, diffChemin
```

Q22. Expliquer le fonctionnement de la boucle itérative comprise entre les lignes 15 à 19 et en déduire le nom du type d'algorithme utilisé.

Q23. Donner ce que renvoie l'instruction `mystere2(G, "a", "f")`. On ne demande pas d'indiquer toutes les étapes de l'algorithme.

Q24. Justifier si ce programme permet au randonneur de trouver le chemin de difficulté cumulée minimale.

III.2 - Deuxième idée : l'algorithme de Dijkstra

Pour résoudre son problème, le randonneur décide d'appliquer l'algorithme de Dijkstra. Il réalise ainsi une fonction `dijkstra` qui prend en arguments :

- la représentation du graphe sous forme de dictionnaire de dictionnaires ;
- le sommet de départ sous forme d'une chaîne de caractères ;
- le sommet d'arrivée sous forme d'une chaîne de caractères ;

et renvoie un dictionnaire dont les clés sont les sommets du graphe et les valeurs sont des couples dont :

- la première composante est la distance totale minimale du point de départ au sommet clé ;
- la seconde composante est le sommet précédent dans le graphe qui permet de réaliser la distance minimale entre le point de départ et le sommet clé.

L'implémentation rappelée ci-dessous de l'algorithme de Dijkstra utilise les quatre variables :

- `aVisiter` : liste des sommets qui doivent être visités ;
- `dejaVisites` : liste des sommets déjà visités ;
- `distance` : le dictionnaire qui sera renvoyé par la fonction ;
- `sTraite` : sommet dont on étudie les voisins pour mettre à jour les distances au point de départ.

La méthode `L.remove(elt)` permet de supprimer la première apparition de `elt` dans la liste `L`.

```
1  def cherche_min(dico:dict, liste:list) -> str:
2      d = float("inf") # Initialisation
3      for s in liste: # parcours des elements de la liste
4          if s in dico and dico[s][0] < d: # recherche de la valeur
               minimale
5              d = dico[s][0]
6              sTraite = s
7      return sTraite
8
9  def unpas(graph:dict, s:str, distance:dict, dejaVisites:list, avisiter:
       list) -> None:
10     avisiter.remove(s) # Mise a jour des sommets a visiter
11     dejaVisites.append(s) # Mise a jour des sommets déjà visités
12     for v in graph[s]: # Mise a jour des distances au point de depart
13         if v not in dejaVisites:
14             if v not in avisiter:
15                 avisiter.append(v)
16             ndistance = distance[s][0] + graph[s][v]
17             if v not in distance or ndistance < distance[v][0]:
18                 distance[v] = (ndistance, s)
19
20 def dijkstra(graph:dict, Sd:str, Sf:str) -> dict:
21     aVisiter = [Sd] # Liste des sommets à visiter
22     dejaVisites = [] # Liste des sommets déjà visités
23     distance = {Sd:(0, Sd)} # Dictionnaire des distances
24     sTraite = Sd # Premier sommet a visiter
25     while sTraite != Sf:
26         sTraite = cherche_min(distance, aVisiter)
27         unpas(graph, sTraite, distance, dejaVisites, aVisiter)
28     return distance
```

Q25. On effectue l'appel `dijkstra(G, "a", "f")` où le graphe G est défini dans la **figure 3**. Dans le tableau du **DR** est représenté le contenu de certaines variables de l'algorithme `dijkstra` en fonction de l'étape de l'itération (comme si un `print` était effectué après la ligne 27). À partir des cases déjà remplies, compléter les cases vides du tableau. Lorsque la clé n'est pas définie dans le dictionnaire, la case du tableau contient un X.

Pour reconstruire un chemin qui réalise la distance optimale entre le point de départ `"a"` et le point d'arrivée `"f"`, on utilise le code suivant. On rappelle que la variable globale `G` a été définie précédemment.

```
1  # On applique l'algorithme de Dijkstra :
2  sInit, sFin = "a", "f"
3  distance = dijkstra(G, sInit, sFin)
4
5  # On construit la liste du chemin en partant de la fin
6  s = sFin
7  chemin = [s]
8  while ........:
9      ..........
10     ..........
11 # On remet le chemin dans l'ordre du début vers la fin
12 chemin.reverse()
13
14 print("Un chemin de ", sInit, " à ", sFin, " est : ", chemin)
15 print("La difficulté minimale est de : ", ..............)
```

Q26. Indiquer le contenu des lignes 8, 9, 10 et 15 du code précédent.

Q27. Expliquer comment pourrait être diminué le nombre de tests d'appartenance à une liste, notamment des lignes 14 et 15, de l'algorithme de Dijkstra.

Pour la fin de la sous-partie III.2, on considère le graphe de la **figure 4** où, pour simplifier, toutes les randonnées sont supposées de difficulté 1. On suppose qu'une représentation de ce graphe par un dictionnaire est fournie dans une variable globale `G1` et que la liste des voisins est triée par ordre alphabétique.

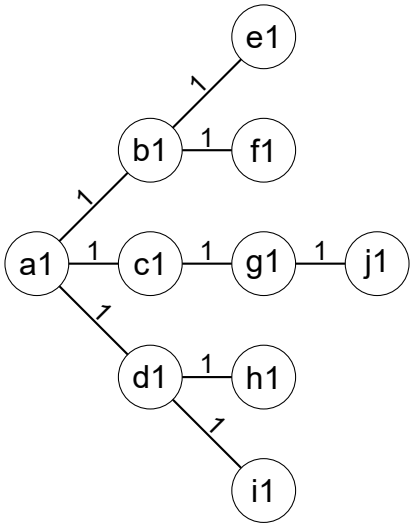

Figure 4 - Graphe `G1`

Q28. Lister les sommets visités par l'appel `dijkstra(G1, "a1", "j1")`, puis par l'appel `dijkstra(G1, "j1", "a1")`.

III.3 - Troisième idée : l'algorithme de Dijkstra bidirectionnel

On constate à l'aide des deux questions précédentes (**Q27** et **Q28**) que, en fonction de la structure du graphe, il est parfois préférable de chercher un chemin qui part du sommet d'arrivée vers le sommet de départ plutôt qu'un chemin qui part du sommet de départ vers le sommet d'arrivée.
L'algorithme de *Dijkstra bidirectionnel* combine ces deux stratégies : au cours de l'algorithme, on va effectuer successivement soit une recherche depuis le point de départ (appelée étape *forward*) soit une recherche depuis le point d'arrivée (appelée étape *backward*). À chaque étape, on choisit ainsi un sommet qui réalise le minimum de la distance soit au sommet de départ, soit au sommet d'arrivée et on y applique l'algorithme de Dijkstra classique.
Précisons l'algorithme. Notons $\mathcal{S}$ l'ensemble des sommets du graphe. Pour tout sommet $i \in \mathcal{S}$, on note $dF(i)$ (respectivement $dB(i)$) la distance minimale actuellement calculée entre le sommet de départ (respectivement d'arrivée) et le sommet i. Ces distances sont actualisées au cours de l'algorithme et valent initialement l'infini. Nous allons maintenir, c'est-à-dire mettre à jour pendant l'exécution de l'algorithme, plusieurs variables :

- `aVisiterF`, `dejaVisitesF` : associées à l'algorithme de Dijkstra partant du point de départ (partie Forward),
- `aVisiterB`, `dejaVisitesB` : associées à l'algorithme de Dijkstra partant du point d'arrivée (partie Backward),
- `Fmin` = $\min\{dF(i),\ i \in$ `aVisiterF`$\}$: distance minimale du sommet de départ à un sommet non encore visité par l'algorithme forward,
- `Bmin` = $\min\{dB(i),\ i \in$ `aVisiterB`$\}$: distance minimale du sommet d'arrivée à un sommet non encore visité par l'algorithme backward,
- `BFmin` = $\min\{dF(i) + dB(i),\ i \in \mathcal{S}\}$: longueur minimale d'un chemin reliant le sommet de départ au sommet d'arrivée,
- `distanceF` (respectivement `distanceB`) : dictionnaire dont les clés sont les sommets et les valeurs sont des couples dont la première composante est la plus petite distance d'un chemin qui relie le sommet depuis le point de départ (respectivement depuis le point d'arrivée) et la seconde est le sommet précédent dans un chemin qui réalise cette distance.

À chaque étape :

- Si `BFmin` $\leq$ `Fmin` + `Bmin`, l'algorithme termine : tout chemin qui réalise le minimum `BFmin` est un chemin optimal.
- Sinon,
  - si `Fmin` $<$ `Bmin`, on choisit un sommet qui réalise `Fmin` et on applique une étape forward de Dijkstra depuis ce sommet, c'est-à-dire qu'on appelle la fonction `unpas` avec les variables forward en mettant à jour les variables `dejaVisitesF`, `aVisiterF` et `distanceF`,
  - si `Bmin` $<$ `Fmin`, on choisit un sommet qui réalise `Bmin` et on applique une itération de l'algorithme de Dijkstra backward depuis ce sommet, c'est-à-dire qu'on appelle la fonction `unpas` avec les variables backward en mettant à jour les variables `dejaVisitesB`, `aVisiterB` et `distanceB`,
  - si `Bmin` $=$ `Fmin`, on choisit un sommet qui réalise ce minimum dans l'ensemble qui contient le moins d'éléments parmi `aVisiterF` et `aVisiterB` et on réalise une étape de Dijkstra à partir de ce sommet. Dans le cas où les deux ensembles contiennent le même nombre d'éléments, on préférera une recherche forward.

- On actualise `Fmin`,`Bmin` et `BFmin` en parcourant l'ensemble des sommets pour lesquels un chemin entre ce sommet et les sommets d'arrivée et de départ a déjà été calculé.

Q29. Compléter la dernière ligne des tableaux du **DR** qui recense les étapes successives de l'algorithme de Dijkstra bidirectionnel sur le graphe G de la **figure 3** avec `a` comme sommet de départ et `f` comme sommet d'arrivée. À chaque étape, on précise si le sommet est visité par la recherche forward (F) ou par la recherche backward (B). Dans chaque case sont précisées les valeurs de `distanceF` et `distanceB`. Pour simplifier la lisibilité, le tableau est découpé en deux parties.

Q30. Justifier si renvoyer la quantité `BFmin` dès qu'un sommet a été atteint par les recherches forward et backward permet de trouver le chemin de longueur minimale.

On donne une implémentation partielle de la fonction `dijkstra_bidirectionnel` qui, étant donné un graphe `graph`, un sommet de départ `Sd` et un sommet final `Sf`, renvoie la distance minimale reliant `Sd` à `Sf` en utilisant l'algorithme de Dijkstra bidirectionnel.

```
1  def dijkstra_bidirectionnel(graph:dict, Sd:str, Sf:str) -> float:
2      aVisiterF = [Sd] # Liste des sommets à visiter Forward
3      dejaVisitesF = [] # Liste des sommets déjà visités Forward
4      aVisiterB = ... # Liste des sommets à visiter Backward
5      dejaVisitesB = ... # Liste des sommets déjà visités Backward
6      # Dictionnaire des distances Forward
7      distanceF = {s:(float('inf'),Sd) for s in graph}
8      # Dictionnaire des distances Backward
9      distanceB = {s:(float('inf'),Sf) for s in graph}
10     distanceF[Sd] = (0, Sd)
11     distanceB[Sf] = (0, Sf)
12     Fmin, Bmin, BFmin = 0, 0, float("inf")
13     while .....:
14         if Fmin < Bmin or \
15                 (Fmin == Bmin and len(aVisiterF) <= len(aVisiterB)):
16             sTraite = cherche_min(distanceF, aVisiterF)
17             unpas(graph, sTraite, distanceF, dejaVisitesF, aVisiterF)
18         else:
19             sTraite = cherche_min(distanceB, aVisiterB)
20             unpas(graph, sTraite, distanceB, dejaVisitesB, aVisiterB)
21         Fmin = min([.......[v][0] for v in aVisiterF if v in distanceF])
22         Bmin = min([.......[v][0] for v in aVisiterB if v in distanceB])
23         L = [..................... for v in distanceB if v in distanceF]
24         if L == []:
25             BFmin = float("inf")
26         else:
27             BFmin = min(L)
28     return BFmin
```

Q31. Compléter la fonction `dijkstra_bidirectionnel` qui, étant donné un graphe `graph`, un sommet de départ `Sd` et un sommet final `Sf`, renvoie la distance minimale reliant `Sd` à `Sf` en utilisant l'algorithme de Dijkstra bidirectionnel. Indiquer précisément le contenu des lignes 4, 5, 13, 21, 22 et 23.
La fonction `min(L:list)` renvoie l'élément minimal d'une liste `L`.

Pour tout couple de sommets a et b du graphe, on note $d(a, b)$ la distance minimale d'un chemin qui relie a à b. On admet que l'algorithme de Dijkstra classique est correct et que, à chaque étape, pour tous les sommets `s` de `dejaVisites`, la valeur $d(a, s)$ est égale à la distance calculée `distance[s][0]`. De plus, tous les sommets non visités sont à une distance de a supérieure à la plus grande des distances déjà calculées.

On s'intéresse maintenant à la correction partielle de l'algorithme de Dijkstra bidirectionnel. Supposons alors par l'absurde que l'algorithme de Dijsktra bidirectionnel a terminé mais que la distance `BFmin` ne soit pas la plus petite distance reliant le sommet de départ s au sommet d'arrivée t. Alors, il existe un chemin $s = s_0, s_1, ..., s_n = t$ qui est de distance `d` strictement inférieure à `BFmin`. Soit s_i un sommet de ce chemin.

Q32. Montrer que s_i appartient soit à `dejaVisitesF` soit à `dejaVisitesB`.

Q33. En déduire qu'il existe un indice i_0 tel que s_{i_0} a été visité par la recherche forward et s_{i_0+1} l'a été par la recherche backward.

Q34. En déduire la correction partielle de l'algorithme de Dijkstra bidirectionnel.

FIN