Exercice 3 (8 points)

Cet exercice porte sur la programmation Python et le routage RIP.

On étudie dans cet exercice des robots qui peuvent se déplacer et communiquer entre eux.

Partie A : Gestion des déplacements du robot

Chaque robot peut se déplacer en avant et tourner de 90 degrés à droite ou à gauche. Il reçoit ses instructions sous la forme d'une chaîne de caractères contenant des lettres et éventuellement des entiers et des parenthèses. S'il lit un 'A', le robot avance d'une distance correspondant à "un pas" (valeur fixée dans les réglages du robot). Il tourne à droite s'il lit un 'D' et à gauche s'il lit un 'G'.

Insérer un entier dans la chaîne permet de répéter une action ou une suite d'actions écrites entre parenthèses.

'12A' est équivalent à 'AAAAAAAAAAAA', '3(AD)' est équivalent à 'ADADAD'.

La figure 1 représente le parcours du robot lorsqu'il reçoit la chaîne '3ADA'

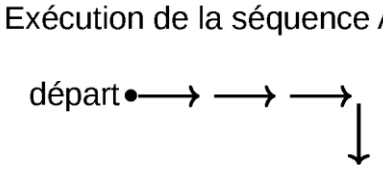


Figure 1. Exemple de parcours

1. Représenter le parcours effectué par le robot s'il reçoit la chaîne '4(AG)'.

Les premières fonctions exécutées vérifient que la chaîne transmise au robot ne comporte pas d'erreurs, en particulier que :

- tous les caractères de la chaîne sont des caractères inclus dans '0123456789ADG()' ;
- on n'a pas d'entier en fin de chaîne, ni d'entier juste avant une parenthèse fermante ;
- le parenthésage est correct (chaque parenthèse ouverte doit être refermée, dans le bon ordre).

Dans toute cette partie, l'opérateur `in` teste la présence ou l'absence d'un élément dans une chaîne de caractères. On peut également le combiner avec l'opérateur `not`.

Exemples :

```
>>> 'a' in 'bateau'
True
```

```
>>> 'y' in 'bateau'
False
>>> 'a' not in 'bateau'
False
>>> 'y' not in 'bateau'
True
```

2. Recopier et compléter la ligne 4 du code de la fonction `caracteres_valides` qui prend en paramètre une chaîne de caractères et renvoie un booléen indiquant si tous les caractères de la chaîne sont valides.

```
1 def caracteres_valides(chaine):
2     valides = '0123456789ADG()'
3     intrus = [c for c in chaine if c not in valides]
4     ... # Vrai si la liste est vide, faux sinon
```

3. Recopier et compléter les lignes 4, 8 et 9 du code de la fonction `entiers_valides` qui prend en paramètre une chaîne de caractères et renvoie un booléen indiquant si tous les entiers sont bien placés dans la chaîne.

```
1 def entiers_valides(chaine):
2     chiffres = '0123456789'
3     if chaine[len(chaine)-1] in chiffres:
4         ...
5     for indice in range(1,len(chaine)):
6         if chaine[indice] == ')':
7             if chaine[indice - 1] in chiffres:
8                 ...
9     ...
```

Pour déterminer si le parenthésage est correct, on propose l'algorithme suivant :

- on initialise une variable `parenthese` à la valeur 0 ;
- on parcourt les caractères de `chaine`, et pour chaque caractère lu,
  - si le caractère lu est `'('`, on augmente `parenthese` de 1 ;
  - si le caractère lu est `')'`, on diminue `parenthese` de 1 ;
  - si `parenthese` est strictement négatif alors la fonction renvoie `False` ;
- si le parcours s'achève, la fonction renvoie `True` si `parenthese` vaut 0 et `False` sinon.

4. Écrire le code de la fonction `parenthesage_correct` qui prend en paramètre une chaîne de caractères et qui renvoie un booléen indiquant si le parenthésage de la chaîne de caractères est correct en utilisant l'algorithme décrit ci-dessous.

La fonction `lire_nombre` prend en paramètres une chaîne de caractères `chaine` et un indice `indice` et renvoie la valeur du nombre dont l'écriture commence à `indice` ainsi que l'indice de son dernier chiffre.

```
1 def lire_nombre(chaine, indice):
2     chiffres = "0123456789"
3     nombre = ''
4     while chaine[indice] in chiffres:
5         nombre = nombre + chaine[indice]
6         indice = indice + 1
7     return (int(nombre), indice-1)
```

Exemple :

```
>>> lire_nombre('AD179AGA', 2)
(179, 4)
```

5. Lors de cet appel, la boucle `while` réalise trois itérations. Donner les valeurs des variables `nombre` et `indice` à la fin de chacune de ces itérations.

La fonction `lire_bloc` prend en paramètres une chaîne de caractères et l'indice d'une parenthèse ouvrante. Elle renvoie le contenu du bloc délimité par cette parenthèse et la parenthèse fermante associée ainsi que l'indice de cette parenthèse fermante.

Exemples :

```
>>> lire_bloc('2(AD)A', 1)
('AD', 4)
>>> lire_bloc('2(AD)3(2AGA)', 6)
('2AGA', 11)
>>> lire_bloc('2(3(AD)G)2A', 1)
('3(AD)G', 8)
```

6. Recopier et compléter les lignes 7, 8 et 9 de la fonction `lire_bloc` donnée ci-dessous.

```
 1 def lire_bloc(chaine, indice):
 2     indice = indice + 1
 3     caractere = chaine[indice]
 4     bloc = ""
 5     compteur = 1
 6     while compteur > 0:
 7         bloc = ...
 8         indice = ...
 9         caractere = ...
10         if caractere == '(':
11             compteur = compteur + 1
12         if caractere == ')':
13             compteur = compteur - 1
14     return (bloc, indice)
```

On suppose que la fonction `execute_mouvement` est déjà implémentée. Cette fonction prend en paramètre un caractère (`A`, `D` ou `G`) et exécute l'action du robot correspondante.

La fonction récursive `lire_parcours` permet de déplacer le robot suivant la chaîne de caractères donnée en paramètre. On donne ci-dessous le code incomplet de cette fonction.

```
 1 def lire_parcours(chaine):
 2     chiffres = "0123456789"
 3     indice = 0
 4     nombre = 1
 5     while indice < len(chaine):
 6         car_lu = chaine[indice]
 7         if car_lu in 'AGD':       # commande simple
 8             for k in range(nombre):
 9                 execute_mouvement(car_lu)
10             nombre = 1
11         elif car_lu in chiffres:  # répétition
12             t = ...
13             nombre = t[0]
14             indice = t[1]
15         elif car_lu == '(':       # début d'un bloc
16             t = ...
17             bloc = t[0]
18             indice = t[1]
19             for k in range(nombre):
20                 ...
21             nombre = 1
22         indice = indice + 1
```

7. Recopier et compléter les lignes 12, 16 et 20 du code de la fonction `lire_parcours`.

Partie B : Communication et routage

On considère un réseau de robots. Chaque robot peut communiquer par radio avec les robots qui se trouvent à proximité.

Si deux robots sont assez proches l'un de l'autre, ils peuvent communiquer directement. Lorsque deux robots sont trop éloignés l'un de l'autre, le message est relayé de robot en robot jusqu'à son destinataire.

On représente un réseau de robots par un graphe dont les arêtes symbolisent la possibilité de communiquer entre deux robots.

Exemple :

Dans la configuration ci-dessous, si le robot numéro 4 veut envoyer un message au robot 12, il devra le transmettre au robot 46 qui le transmettra alors au robot 12.

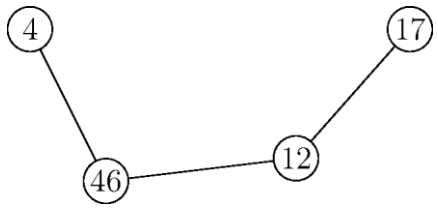


Figure 2. Un réseau constitué de quatre robots

Le routage des messages est effectué à l'aide du protocole RIP qui minimise la distance calculée en nombre de sauts réalisés entre deux routeurs. Chaque robot joue le rôle d'un routeur : il maintient une table de routage dans laquelle figurent les informations permettant de joindre les autres robots connus.

Une table de routage contient trois colonnes :

- la colonne "destination" donne l'identifiant du robot à atteindre ;
- la colonne "prochain robot" donne l'identifiant du prochain robot sur le chemin vers la destination ;
- la colonne "distance" indique le nombre total de transmissions à effectuer pour joindre la destination.

Exemple :

Le réseau suivant est constitué par les robots 46, 4, 57, 91, 22 disposés comme dans le graphe ci-dessous.

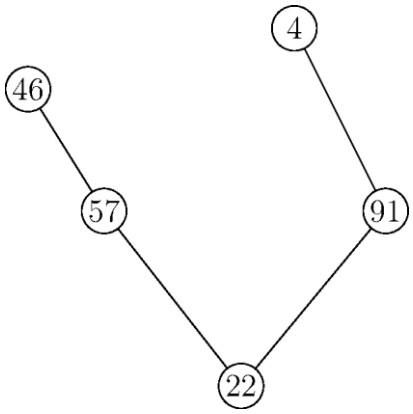


Figure 3. Réseau constitué par les cinq robots

La table de routage du robot 91 est :

destination	prochain robot	distance
4	4	1
46	22	3
22	22	1
57	22	2

On suppose maintenant qu'à la suite d'un déplacement du robot 4, une liaison directe avec le robot 46 est possible :

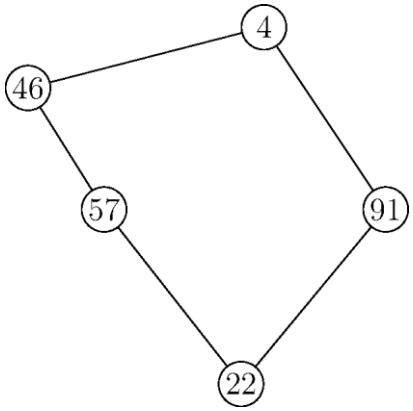


Figure 4. Le même réseau après mouvement du robot 4

8. Indiquer les modifications apportées à la table de routage du robot 91.

Un nouveau robot, portant l'identifiant 87, rejoint le réseau en contactant le robot numéro 91. Il lui envoie sa table de routage :

Table de routage du robot 87 :

destination	prochain robot	distance
91	91	1
63	63	1
36	63	2

9. Décrire les nouvelles modifications à apporter à la table de routage du robot 91.

Partie C : Programmation du routage

Afin de gérer la communication entre les robots du réseau, on implémente la classe `Module_comm` et chaque robot possède une instance de cette classe.

```
1 class Module_comm:
2     def __init__(self, id_robot):
3         self.identifiant = id_robot
4         self.table_routage = {}
```

Chaque clé du dictionnaire `table_routage` est l'identifiant d'un robot du réseau. La valeur associée est un dictionnaire donnant le premier robot sur le chemin vers ce robot ainsi que le nombre total de transmissions c'est-à-dire le nombre de sauts à effectuer.

La table du routage du robot 91 de la figure 3 est implémentée comme suit :

```
1 {
2     4  : { "prochain": 4, "distance": 1},
3     46 : { "prochain": 22, "distance": 3},
4     22 : { "prochain": 22, "distance": 1},
5     57 : { "prochain": 22, "distance": 2},
6 }
```

La méthode `ajouter_voisin` prend en paramètre l'identifiant d'un robot avec lequel il peut directement communiquer puis crée une entrée dans le dictionnaire `table_routage`.

10. Écrire le code de la méthode `ajouter_voisin`.

La méthode `nombre_sauts` prend en paramètre l'identifiant d'un robot et renvoie le nombre de sauts nécessaires pour communiquer avec ce robot. Pour cette implémentation, on fixe le nombre de sauts maximum à 15.

Si l'identifiant reçu par `nombre_sauts` ne figure pas dans la table de routage, la fonction renvoie la valeur 16.

On rappelle que les opérateurs `in` et `not in` permettent de savoir si un objet figure dans les clés d'un dictionnaire Python.

Exemple :

```
>>> D = {4  : { "prochain": 31, "distance": 2},
         46 : { "prochain": 22, "distance": 3} }
>>> 4 in D
True
>>> 5 not in D
True
```

11. Recopier et compléter la ligne 5 du code de la méthode `nombre_sauts` :

```
1     def nombre_sauts(self, identifiant):
2         if identifiant not in self.table_routage:
3             return 16
4         else:
5             return ...
```

12. Écrire le code de la méthode `voisins` qui renvoie la liste de tous les robots avec lesquels il peut directement communiquer.

Afin de ne pas communiquer d'informations inutiles, on souhaite communiquer à un robot voisin l'extrait de la table de routage qui contient uniquement les lignes pour lesquelles ce robot voisin n'est lui-même pas le prochain robot sur la route.

La méthode `communiquer_extrait_table` prend en paramètre l'identifiant d'un robot `voisin` et communique cet extrait de la table de routage. L'extrait de table renvoyé est de type dictionnaire et il possède la même structure que la table de routage complète.

13. Écrire le code de la méthode `communiquer_extrait_table`.