Exercice 3 (8 points)

Cet exercice porte sur les protocoles de routage, la programmation orientée objet et la notion de file.

Le but de cet exercice est de simuler la façon dont des routeurs construisent leur table de routage selon le protocole RIP.

On considère à titre d'exemple le réseau ci-dessous :

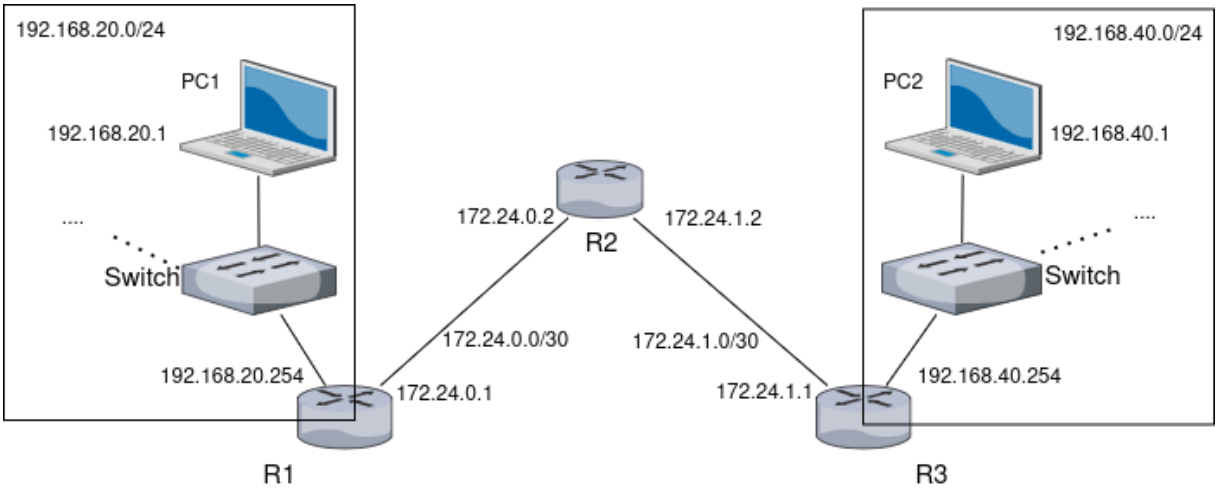


Figure 1. Exemple de réseau

Les adresses des quatre réseaux sont notées sous la forme `IP/S` où `IP` est une adresse IP donnée par quatre octets écrits en base 10 et `S` est un entier inférieur à 32 appelé *suffixe*. Une adresse IP peut être écrite sur 32 bits en binaire. Le suffixe indique que les `S` premiers bits de l'adresse correspondent à la partie fixe de l'adresse et les suivants à la partie propre à la machine.

Pour que deux machines soient sur un même réseau local avec un suffixe `S`, il faut que les `S` premiers bits de leurs adresses IP soient identiques.

Partie A

1. Identifier la partie fixe des adresses IP des machines branchées dans le réseau `192.168.20.0/24`.

2. Dans chaque réseau, il y a deux adresses réservées. Donner le nombre maximum de machines qui peuvent être branchées dans un réseau dont le suffixe est 30.

Le protocole RIP permet de minimiser la distance, c'est-à-dire le nombre de routeurs traversés par un paquet de données. Voici la table de routage du routeur R1 dans le réseau représenté sur la figure 1, selon le protocole RIP :

Table de routage de R1			
Destination	Interface	Passerelle	Distance
192.168.20.0/24	192.168.20.254	–	0
172.24.0.0/30	172.24.0.1	–	0
172.24.1.0/30	172.24.0.1	172.24.0.2	1
192.168.40.0/24	172.24.0.1	172.24.0.2	2

3. Donner la table de routage du routeur R2 pour le réseau représenté sur la figure 1, selon le protocole RIP.

Dans les parties B et C, les adresses sont représentées en Python par des tuples de 4 entiers. Par exemple, l'adresse `192.168.40.1` est représentée par `(192, 168, 40, 1)`.

Partie B

Afin de comparer les adresses IP pour savoir si elles appartiennent à un même réseau, on calcule le *masque de réseau* à partir du suffixe. Les premiers bits, correspondant à la partie fixe, sont mis à 1 et les autres sont mis à 0.

4. Donner le tuple d'entiers correspondant à un masque pour un réseau dont le suffixe est 30.

Pour isoler la partie fixe d'une adresse IP, on utilise le ET bit à bit. Cela consiste à appliquer l'opération booléenne ET à tous les bits se trouvant à la même position dans les deux nombres écrits en binaire.

```
   01110110  (118)
ET 11010010  (210)
-----------
   01010010   (82)
```

En Python, il est possible de faire cette opération en utilisant l'opérateur `&` directement entre deux nombres écrits en base 10 :

```
>>> 118 & 210
82
```

5. Expliquer pourquoi `255 & A` est égal à `A`, si `A` est un entier compris entre 0 et 255.

6. Déterminer, en justifiant la réponse, si `(172, 20, 189, 8)` et `(172, 20, 191, 203)` sont dans un même réseau dont le masque est `(255, 255, 252, 0)`.

La fonction `meme_reseau` prend en paramètres trois tuples de quatre entiers `adresse1`, `adresse2` et `masque`, et renvoie un booléen indiquant si `adresse1` et `adresse2` sont dans le même réseau en utilisant le masque de réseau `masque`.

```
def meme_reseau(adresse1, adresse2, masque):
    for i in range(4):
        octet1 = adresse1[...] & masque[...]
        octet2 = ...
        if ...:
            return ...
    return ...
```

7. Recopier et compléter le code de la fonction `meme_reseau`.

Partie C

Nous allons maintenant définir les classes permettant de simuler un réseau de réseaux locaux.

On dispose d'une classe `File` qui permet d'instancier une file. Cette classe dispose, en plus du constructeur qui renvoie une file vide, des méthodes `enfiler`, `defiler` et `est_vide` : pour une file `file1`,

- `file1.enfiler(e)` ajoute l'élément `e` à la fin de la file `file1` et ne renvoie rien ;
- `file1.defiler()` enlève le premier élément de la file et le renvoie ;
- `file1.est_vide()` renvoie `True` si la file `file1` est vide et `False` sinon.

Chaque réseau local est modélisé à l'aide de la classe `Reseau` dont voici le constructeur :

```
class Reseau:
    def __init__(self, adresse, masque):
        self.adresse = adresse
        self.masque = masque
        self.machines = {}
        self.file = File()
```

Dans cette modélisation, c'est le réseau qui s'occupe de l'acheminement des paquets. Lorsqu'une machine envoie un paquet, il est ajouté à la file du réseau. Pour simplifier, tout paquet envoyé sur un réseau local sera distribué à toutes les machines de ce réseau, à l'exception de celle qui a émis le paquet. Un paquet est un couple de la forme `(orig, donnees)` où `orig` est l'adresse IP de la machine envoyant le paquet et `donnees` correspond aux données envoyées.

Cette classe possède également un méthode `connexion`, non présentée ici, qui prend en paramètre une file et qui permet d'attribuer une nouvelle adresse à une machine, en associant cette adresse à la file dans le dictionnaire `machines` du

réseau concerné puis de renvoyer l'adresse. La méthode `diffuser`, de la classe `Reseau` permet de traiter les paquets contenus dans sa file et de les déposer dans les files des machines destinataires.

```
    def diffuser(self):
        while not ...:
            paquet = ...
            orig, donnees = paquet
            for m in self.machines:
                if ...:
                    self.machines[m].enfiler(...)
```

8. Recopier et compléter le code de la méthode `diffuser`.

La classe `Routeur` permet de définir les routeurs. Ils possèdent une interface pour chaque réseau auquel ils sont connectés. On utilise pour cela la classe `Interface`. Chaque interface d'un routeur est identifiée par son indice dans le tableau `interfaces`. On donne ci-dessous une partie du code de ces deux classes :

```
class Interface:
    def __init__(self, reseau):
        self.sortie = reseau.file
        self.file = File()
        self.adresse = reseau.connexion(self.file)
        self.masque = reseau.masque

class Routeur:
    def __init__(self, nom):
        self.nom = nom
        self.interfaces = []
        self.table = {}

    def branchement(self, reseau):
        k = len(self.interfaces)
        self.table[reseau.adresse] = (k, None, 0)
        self.interfaces.append(Interface(reseau))
```

Les routeurs communiquent entre eux en envoyant des *vecteurs de distance* qui sont des couples (réseau, distance). Par exemple, le routeur R1 du réseau représenté sur la figure 1 envoie à ses voisins les vecteurs `((192, 168, 20, 0), 0)`, `((172, 24, 0, 0), 0)`, `((172, 24, 1, 0), 1)` et `((192, 168, 40, 0), 2)`.

La construction des tables de routage, selon RIP, repose sur l'algorithme de Bellman-Ford :

- Initialement, la table de chaque routeur ne contient que les réseaux locaux auxquels il est connecté, qui sont à la distance 0.
- Lorsque les vecteurs d'un voisin sont reçus :

- Si la table du voisin contient une route vers un nouveau réseau local, ce réseau est ajouté à la table.
- Si elle contient une route vers un réseau déjà connu et que cette nouvelle route est plus courte, la table est mise à jour.
- Si elle contient une route vers un réseau qui n'améliore pas le chemin actuel, la table n'est pas modifiée.

Les tables de routage seront représentées sous la forme de dictionnaires associant à chaque adresse réseau un triplet `(i_interf, passerelle, distance)`, où `i_interf` est l'indice de l'interface dans le tableau `interfaces` du routeur. Lorsque le routeur est directement connecté au réseau local, on met `None` à la place de la passerelle.

La méthode `m_a_j` de la classe `Routeur` prend trois paramètres, en plus de `self` :

- `passerelle` qui est l'adresse de l'interface du routeur qui envoie ses vecteurs ;
- `i_interf` qui est l'indice dans `self.interfaces` de l'interface concernée du routeur dont on fait la mise à jour ;
- `vecteurs` qui est le tableau des vecteurs de distance reçus.

Cette méthode met à jour la table de routage selon l'algorithme de Bellman-Ford. Si la table a été modifiée suite à cette mise à jour, le routeur utilise la méthode `envoyer_vecteurs` qui prend en paramètre, en plus de `self`, un entier `i_interf` qui correspond à l'interface par laquelle il a reçu les informations et qui envoie les vecteurs sur toutes les autres interfaces. Cette méthode sera étudiée par la suite.

Prenons l'exemple d'une instance de la classe `Routeur`, qui serait contenue dans une variable `routeur_m` :

```
>>> routeur_m.table
{(172, 64, 0, 0): (1, None, 0),
 (172, 60, 0, 0): (0, (172, 58, 0, 2), 1),
 (192, 168, 47, 0): (0, (172, 58, 0, 2), 2),
 (172, 62, 0, 0): (0, (172, 58, 0, 2), 2)}
```

On considère également le tableau de vecteurs suivant :

```
vecteurs = [((192, 168, 72, 0), 0), ((172, 62, 0, 0), 0),
            ((172, 64, 0, 0), 0), ((172, 60, 0, 0), 1),
            ((192, 168, 47, 0), 1)]
```

9. Déterminer la valeur de `routeur_m.table` après l'appel `routeur_m.m_a_j((172, 64, 0, 2), 1, vecteurs)`.

10. Recopier et compléter les lignes 4 et 6 du code ci-après de la méthode `m_a_j`.

```
1     def m_a_j(self, passerelle, i_interf, vecteurs):
2         a_ete_modif = False
3         for adresse, d in vecteurs:
4             if ... or ...:
5                 self.table[adresse] = (i_interf,
passerelle, d+1)
6                 a_ete_modif = ...
7         if a_ete_modif:
8             self.envoyer_vecteurs(i_interf)
```

La méthode `vecteurs` de la classe `Routeur` permet de générer, pour le routeur considéré, le tableau des vecteurs de distance à envoyer via une interface dont l'indice `i` est donné. Pour éviter de propager des erreurs ou de générer des cycles, selon le protocole RIP, les vecteurs ne contiennent pas les routes qui passent par l'interface sur laquelle ils sont envoyés.

```
1     def vecteurs(self, i):
2         res = []
3         for adresse in self.table:
4             i_interf, passerelle, dist = self.table[adresse]
5             if ...:
6                 res.append(...)
7         return res
```

11. Recopier et compléter les lignes 5 et 6 du code de la méthode `vecteurs`.

On rappelle que lorsque les vecteurs reçus sur une interface `i_interf` entraînent une modification de la table de routage d'un routeur, ce routeur envoie ses vecteurs sur toutes les autres interfaces. La méthode `envoyer_vecteurs` ci-après prend en paramètre, en plus de `self`, un entier `i_interf` et envoie les vecteurs sur toutes les autres interfaces.

```
1     def envoyer_vecteurs(self, i_interf):
2         for i in range(len(self.interfaces)):
3             if ...:
4                 adresse = self.interfaces[i].adresse
5                 v = self.vecteurs(...)
6                 paquet = (adresse, v)
7                 ... .enfiler(paquet)
```

12. Recopier et compléter les lignes 3 à 7 du code de la méthode `envoyer_vecteurs`.

La méthode `traitement` de la classe `Routeur` parcourt toutes les files des interfaces et traite les paquets qu'elles contiennent.

```
1     def traitement(self):
2         for i in range(len(self.interfaces)):
3             file = self.interfaces[i].file
4             while not file.est_vide():
```

```
5                   orig, donnees = file.defiler()
6                   self.m_a_j(orig, i, donnees)
7
```

Afin de simuler la construction des tables de routage, nous allons procéder de la manière suivante, une fois les routeurs connectés aux différents réseaux :

- Chaque routeur va envoyer ses vecteurs initiaux sur toutes ses interfaces (on utilise la valeur `-1` pour le paramètre `i_interf` pour ne sauter aucune interface). Les routeurs sont tous contenus dans le tableau `routeurs`.
- On diffuse les paquets se trouvant dans toutes les files des réseaux (qui sont tous contenus dans le tableau `reseaux`).
- On utilise la méthode `traitement` de tous les routeurs afin de mettre à jour leurs tables et éventuellement de diffuser à nouveaux des vecteurs.
- S'il n'y a plus de paquets en attente dans les réseaux, alors on a fini.

13. Recopier et compléter la suite du script ci-dessous à partir de la ligne 4 afin de construire les tables de routage des routeurs contenus dans le tableau `routeurs` suivant l'algorithme ci-dessus.

```
1 for routeur in routeurs:
2     routeur.envoyer_vecteurs(-1)
3
4 continuer = True
5 while continuer:
6     ...
```