Algorithmes et Pensée Computationnelle

Algorithmes probabilistes

Le but de cette séance est de comprendre les algorithmes probabilistes. Ceux-ci permettent de résoudre des problèmes complexes en relativement peu de temps. La contrepartie est que le résultat obtenu est généralement une solution approximative du problème initial. Néanmoins, ces algorithmes demeurent très utiles pour beaucoup d'applications.

1 Monte-Carlo

Question 1: (*10 minutes*) **Un jeu de hasard : Python**
Supposez que vous lanciez une pièce de monnaie **l** fois et que vous voulez calculez la probabilité d'avoir un certain nombre de piles. Vous devez programmer un algorithme probabiliste, permettant de calculer cette probabilité. Pour ce faire, vous devez compléter la fonction *proba(n,l,iter)* contenue dans le fichier `Piece.py` (Dans le dossier `Code`). La fonction *Piece(l)* permet de créer une liste contenant des 0 et des 1 aléatoirement avec une probabilité $\frac{1}{2}$. Considérez un chiffre 1 comme une réussite (pile) et 0 comme un échec (face).

Conseil

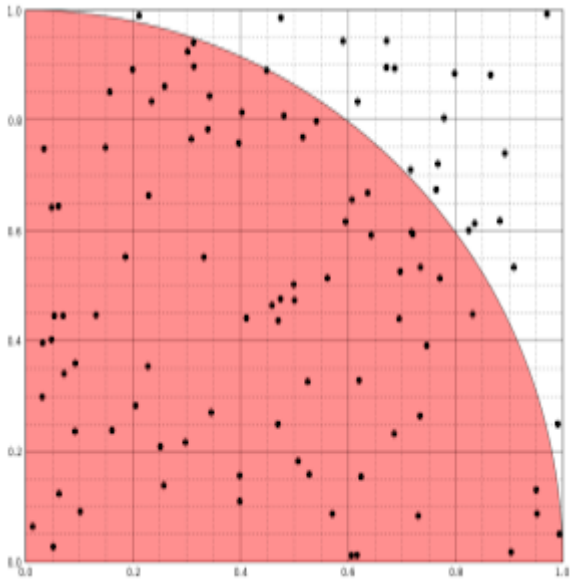
1. Pour estimer empiriquement la probabilité d'un événement, comptez le nombre de fois que l'événement en question se produit en effectuant un nombre d'essais. Puis divisez le nombre d'occurrences de l'événement par le nombre total d'essais. Par exemple, si vous voulez estimer la probabilité d'obtenir un 2 avec un dé. Lancez le dé 1000 fois, comptez le nombre de fois que vous obtenez 2, et divisez le résultat par 1000.
2. Pour choisir de façon aléatoire un nombre dans un intervalle, vous pouvez utiliser la méthode `random.randint(début, fin)` après avoir importé le module `random`.

Solution

```python
import random


# La fonction Piece retourne une liste contenant des 0 et des 1, considérez un 1 comme un succès, i.e. une fois où la pièce tombe sur pile, et 0 comme un échec
def Piece(l):
    return [random.randint(0, 1) for i in range(l)]


def proba(n, l, iter=10000):
    # n correspond au nombre de succès et l au nombre d'essais. Iter correspond au nombre d'expérience que vous allez
    # réaliser pour obtenir la réponse. Cela devrait être grand mais pas trop (sinon le programme prendra trop de
    # temps à s'exécuter). 10000 est un bon nombre d'itérations.
    proba = 0
    for i in range(iter):
        temp = Piece(l)  # On simule une expérience de l lancés.
        count = sum(temp)  # On compte le nombre de fois que l'on obtient pile
        if count == n:  # Si le nombre de pile obtenu correspond à la probabilité que l'on veut estimer
            proba += 1  # On ajoute 1 à notre estimateur de probabilité
    return proba / iter  # Divise notre estimateur de probabilité par le nombre total d'expériences réalisées.


n = 5
l = 10
print("La probabilité d'avoir {} pile en {} lancés de pièce est approximativement égale à {}".format(n, l, proba(n, l,10000)))
```

Question 2: (*15 minutes*) **Une approximation de π : Python**

L'objectif de cet exercice est d'écrire un algorithme probabiliste permettant d'approximer le nombre π. Imaginez un plan sur lequel $0 < x < 1$ et $0 < y < 1$. Sur ce dernier, nous allons dessiner un quart de cercle centré en (0,0) et avec un rayon de 1. Par conséquent, un point dans cet espace se trouve à l'intérieur ou sur le cercle si $x^2 + y^2 <= 1$. Vous trouverez ci-dessous une illustration de la situation :



La première étape de cet exercice consiste à créer une fonction permettant de déterminer si un point est à l'intérieur (zone rouge) ou à l'extérieur du cercle. Puis, générez 10000 points dans cet espace (x et y devraient appartenir à l'intervalle [0,1]). Pour ce faire, vous pouvez utliser la fonction *random.random()* après avoir importé le module **random**. Vous pouvez obtenir l'approximation de π à partir de la formule suivante : $\pi \approx (\frac{\text{Nombre de points dans le cercle}}{\text{Nombre total de points}}) \cdot 4$. Votre réponse devrait être assez proche du vrai chiffre π.

Conseil

La fonction `random.random()` génère aléatoirement un chiffre compris entre 0 et 1. Etant donné que vous devez simuler des points en 2 dimensions, vous devrez utiliser 2 fois cette fonction.

Solution

```
1  import random
2
3  def inside(point):#Point définit sous la forme d'un tuple
4      # Cette fonction permet de vérifier si un point se trouve à
       l'intérieur ou sur le cercle
5      if (point[0]**2+point[1]**2) <= 1:
6          return 1
7
8      else:
9          return 0
10
11 def app():
12     count = 0 #On initialise le nombre de points dans le cercle
13     iter = 100000000 # Plus cette valeur augmente, plus on se
       rapproche de la valeur de pi
14     for i in range(iter):
15         temp1 = random.random()#Génère la première coordonnée
16         temp2 = random.random()#Génère la deuxième coordonnée
17         temp = (temp1,temp2)#Crée le point
18
19         count += inside(temp)#On appelle la fonction. Si le point est
       dans le cercle, elle retourne 1, par conséquent on ajoute 1 au
       compteur. Sinon elle retourne 0, on ajoute donc rien.
20
21     return count/iter*4#Retourne selon la formule donnée dans
       l'exercice.
22
23 print("L'approximation du chiffre pi est : {}".format(app()))
```

2 Fingerprinting

Question 3: (*15 minutes*) **Fingerprinting : Une mission pour l'agente secrète Alice : Python**
Dans cet exercice, vous prendrez le rôle de l'agente secrète Alice. Cette dernière enquêtait sur la disparition de son collègue, l'agent Bob, et se doutait que l'indice clé qui la mènerait à la vérité se trouvait dans la boîte mail de Bob. Alice arriva à trouver un bout de papier avec écrit dessus : "*Mon mot de passe est l'empreinte de ceciestmonmotdepasse*". Aidez Alice à trouver l'empreinte du mot de passe !

Pour cela, vous devez compléter deux fonctions :

1. `is_a_prime_number(num)` qui vérifie que `num` est un nombre premier ou pas. Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs. Ces deux diviseurs sont 1 et le nombre considéré, puisque tout nombre a pour diviseurs 1 et lui-même, les nombres premiers étant ceux qui n'en possèdent aucun autre.
2. `fingerprinting(p, message)` qui implémente l'algorithme de fingerprinting suivant :
   (a) Si `p` est un nombre premier, calculez la valeur de hachage de la chaîne à l'aide de la fonction `hash(...)`, puis calculez le modulo du résultat du hachage.
   (b) Sinon, affichez un message qui dit que le nombre n'est pas un nombre premier.

Si vous réussissez à implémenter les deux fonctions correctement, le programme vous affichera : `Connexion réussie? True`.
À vos ordis, détectives !

```
1  import base64
2
3  # num est un nombre entier
4  def is_a_prime_number(num):
5      # Partie à compléter
6
7
8  # p est un nombre premier et message est une chaine de caractères
9  def fingerprinting(p, message):
10     # Partie à compléter
11
12
13 # password est une chaine de caractères et your_details est un tuple avec le
14 # format suivant (nombre premier, hash du mot de passe)
15 def login(password, your_details):
16     return your_details[1] % your_details[0] ==
       fingerprinting(your_details[0], password)
17
18
19
20 # Début de votre programme
21 password = "ceciestmonmotdepasse"
22 your_details = (19, hash(password))
23 success = login(password, your_details)
24
25 print("Connexion réussie? " + str(success))
26 if success:
27     message = '''SmUgc2VyYWlzIGNvbmZpbsOpIGNoZXogbWVzIHBhcmVudHMgw
28                  6AgbGEgY2FtcGFnbmUgbGVzIGRldXggcHJvY2hhaW5lIHNlbWF
29                  pbmVzLCBldCBqZSBuJ2F1cmFpcyBwYXMgYWNjw6hzIMOgIEludGV
30                  ybmV0LiDDgCBiaWVudMO0dCE='''
31     print(base64.b64decode(message).decode())
```

Conseil

Pour vérifier si un nombre n est premier, il faut parcourir tous les nombres à partir de 2 à $(n//2+1)$ et vérifier si chaque nombre divise n. Si un nombre qui divise n est trouvé, il faut retourner `False`. Si aucun diviseur est trouvé alors cela signifie que n est premier et il faut retourner `True`.

Solution

```python
import base64

# num est un nombre entier
def is_a_prime_number(num):
    if num<=1:
        return False
    for i in range(2, (num//2)+1):
        if num%i==0:
            return False
    return True

# p est un nombre premier et message est une chaine de caractères
def fingerprinting(p, message):
    if is_a_prime_number(p):
        result = hash(message) % p
        return result
    print(str(p) + " is not a prime number!")

# password est une chaîne de caractères et your details est un tuple avec le
# format suivant (nombre premier, hash du mot de passe)
def login(password, your_details):
    return your_details[1] % your_details[0] == fingerprinting(your_details[0], password)

if __name__ == "__main__":
    password = "ceciestmonmotdepasse"
    your_details = (19, hash(password))
    success = login(password, your_details)

    print("Connexion réussie? " + str(success))
    if success:
        message = '''SmUgc2VyYWlzIGNvbmZpbsOpIGNoZXogbWVzIHBhcmVudHMgw
                     6AgbGEgY2FtcGFnbmUgbGVzIGRldXggcHJvY2hhaW5lIHNlbWF
pbmVzLCBldCBqZSBuJ2F1cmFpcyBwYXMgYWNjw6hzIMOgIEludGV
                     ybmV0LiDDgCBiaWVudMO0dCE='''
        print(base64.b64decode(message).decode())
```

3 Las Vegas

Question 4: (*10 minutes*) **Un point dans un cercle unitaire : Python** Optionnel

Les algorithmes de Monte Carlo sont des algorithmes probabilistes dont la sortie peut être incorrecte avec une certaine probabilité, qui est généralement faible. En revanche, un algorithme de Las Vegas est un algorithme probabiliste qui trouve toujours le bon résultat lorsqu'il existe. Son inconvénient est que sa complexité temporelle ne peut être garantie à l'avance car elle dépend des données passées en paramètres.

L'objectif de cet exercice est de programmer un algorithme probabiliste permettant de donner un point contenu dans un cercle unitaire.

Conseil

Vous pouvez vous inspirer de l'exercice 2 *(Approximation de π)*.

Solution

```
import random


def inside(point):  # Point est défini sous la forme d'un tuple
    # Cette fonction permet de vérifier si un point se trouve à
    l'intérieur du cercle
    if (point[0] ** 2 + point[1] ** 2) < 1:
        return 1

    else:
        return 0


def app():
    for i in range(10000):
        temp1 = random.random()  # Génère la première coordonnée
        temp2 = random.random()  # Génère la deuxième coordonnée
        temp = (temp1, temp2)  # Crée le point

        if (inside(temp)) :
            return temp # Retourne le point trouvé.


print("Voilà un point dans un cercle unitaire  : {}".format(app()))
```

Question 5: (*20 minutes*) **Quicksort - Algorithme de Las Vegas : Python**

Dans cet exercice, vous allez implémenter un algorithme de tri rapide (quicksort) sur une liste d'éléments avec l'algorithme de Las Vegas.

L'algorithme de tri rapide applique un paradigme *divide-and-conquer* afin de trier un ensemble de nombres `A`. Il fonctionne en trois étapes :

1. il choisit d'abord un élément pivot, `A[q]`, en utilisant un générateur de nombres aléatoires (d'où sa nature d'algorithme dit probabiliste) ;
2. puis il réorganise le tableau en deux sous-tableaux $A[p...q-1]$ et $A[q+1...r]$, où les éléments des premier et deuxième tableaux sont respectivement plus petits et plus grands que $A[q]$.
3. L'algorithme applique ensuite récursivement les étapes de tri rapide ci-dessus sur les deux tableaux indépendants, produisant ainsi un tableau entièrement trié.

Complétez le code suivant :

```
import random
```

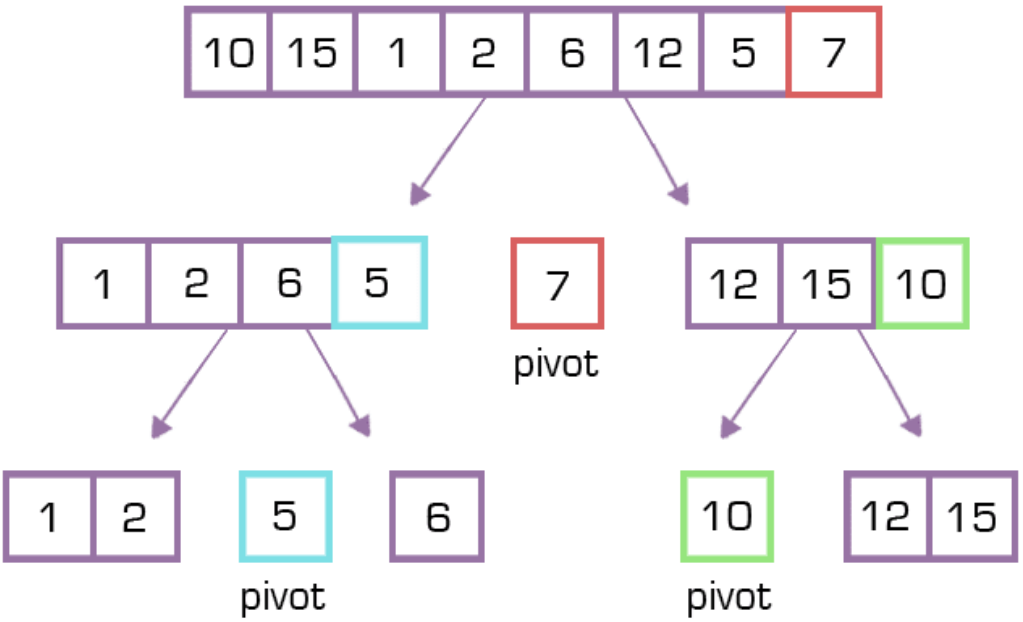


FIGURE 1 – Illustration de l'algorithme de tri rapide

```
2
3  # lst représente la liste à trier, l l'index 0 et r la taille de la liste -1
4  def sort(lst, l, r):
5          # mettre une condition pour arrêter la récursivité
6
7
8          pivot_index = ... # Partie à compléter: Choisissez un pivot compris
       entre 0 et la longueur de votre liste - 1
9
10         # Déplacer votre pivot dans votre liste
11
12         # Partitionnez votre liste de telle sorte que les éléments plus petits
       que le pivot soient placés avant celui-ci et les éléments plus grands
       soient placés après
13
14         # Replacer votre pivot à l'endroit adéquat
15
16         # Effectuez le tri de facon récursive sur les parties gauches et
       droites de la liste
17
18 def quicksort(items):
19     if items is None or len(items) < 2:
20         return
21     sort(items, 0, len(items) - 1)
22
23 l = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
24 quicksort(l)
25 print('Liste triée:    ', l)
```

Conseil

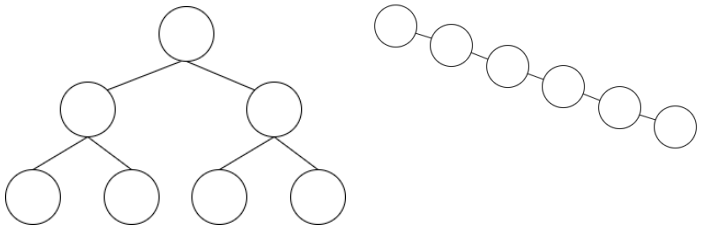
Pour le choix de votre élément pivot, pensez à utiliser la méthode `randint()` de la librairie `random`.

Solution

```python
import random

# lst représente la liste à trier, l l'index 0 et r la taille de la liste -1
def sort(lst, l, r):
        # Dans le meilleur des cas, on arrête la récursivité
        if r <= l:
            return

        # Choix du pivot
        pivot_index = random.randint(l, r)

        # On déplace le pivot au premier élément
        lst[l], lst[pivot_index] = lst[pivot_index], lst[l]

        # partition
        i = l
        for j in range(l+1, r+1):
            if lst[j] < lst[l]:
                i += 1
                lst[i], lst[j] = lst[j], lst[i]

        # On place le pivot à la position adéquate
        lst[i], lst[l] = lst[l], lst[i]

        # On effectue le tri de fa con récursive sur les parties gauches et droites de la liste
        sort(lst, l, i-1)
        sort(lst, i+1, r)

def quicksort(items):
    if items is None or len(items) < 2:
        return
    sort(items, 0, len(items) - 1)

l = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
quicksort(l)
print('Liste triée:   ', l)
```

4 Treap

Question 6: (🕒 *30 minutes*) **Insertion dans un Treap : Python et Papier**
Un arbre binaire de recherche montre de meilleures performances lorsqu'il est équilibré. Ainsi, la complexité d'une opération de recherche, d'insertion ou de suppression d'un nœud dans l'arbre équilibré *a* est de $O(logn)$. Cette complexité est de $O(n)$ dans l'arbre *b*.



(a) Arbre équilibré (b) Arbre non équilibré

FIGURE 2 – Exemple d'arbres binaires de recherche

Afin de s'assurer d'obtenir un arbre binaire de recherche presque équilibré, on peut utiliser les propriétés d'un Heap (ou tas) dont les éléments sont ordonnés en suivant une priorité. Dans un Max-heap (Figure 3), le nœud ayant la priorité maximale se trouve toujours au sommet de l'arbre. Les nœuds parents auront toujours une priorité plus grande que les nœuds enfants. Dans un Min-heap tel que présenté en cours, le nœud ayant la plus faible priorité se trouve au sommet et dans cette configuration, les nœuds parents auront toujours une priorité plus petite que les nœuds enfants.

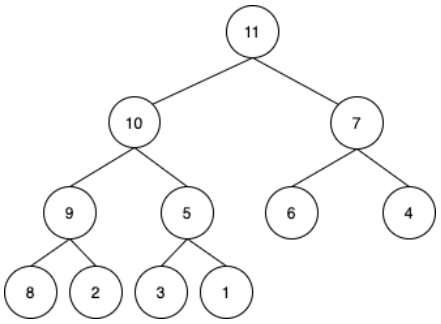


FIGURE 3 – Exemple de Max-heap

Un Treap (ou arbretas) est une fusion entre un arbre binaire de recherche et un Heap. En construisant un Treap, vous devez vous assurer que les deux conditions ci-dessous soient respectées :

- Le nœud de gauche a une valeur plus petite que le nœud parent, tandis que le nœud de droite a toujours une valeur plus grande que le nœud parent. *–propriété d'un arbre binaire de recherche*
- Chaque enfant a une priorité plus petite que la priorité du parent. *–propriété d'un max-heap*

Lors de l'insertion dans un Treap, si un nœud ne respecte pas les propriétés édictées ci-dessus, on procède à une rotation en permutant la position du nœud avec celle de son parent jusqu'à ce qu'on obtienne un Treap.
Soit la liste suivante :

```
liste = [5, 2, 1, 4, 9, 8, 10]
```

Dans le fichier `treap.py` se trouvant sur Moodle, créer une nouvelle liste `nodes` qui contient des ensembles de tuples à deux éléments. Chaque tuple contiendra un élément de `liste` et une valeur aléatoire (comprise entre 0 et 99) représentant une priorité.
Notez ci-dessous les valeurs que vous obtiendrez :

nodes = [(5, ...), (2, ...), (1, ...), (4, ...), (9, ...), (8, ...), (10, ...)]

Placez les éléments de `nodes` dans un Treap. Utilisez l'espace ci-dessous pour dessiner le Treap qui sera obtenu.

Conseil

— Commencez d'abord par construire un arbre binaire de recherche en utilisant uniquement les clés, ensuite utilisez les priorités pour réorganiser les nœuds de votre arbre.
— Une fois complété, le programme `treap.py` se trouvant sur Moodle générera un Treap en tenant compte de la position des éléments.
— Utilisez la fonction `random.randrange(100)` pour générer des nombres aléatoires compris entre 0 et 99.
— La propriété du Heap à satisfaire est que la priorité de la racine doit toujours être plus grande que celle de ses nœuds enfants. Les méthodes `left_rotate` et `right_rotate` permettent de réarranger les nœuds de façon à ce que la propriété du Heap soit satisfaite. Vous pouvez vous référer à l'illustration ci-dessous pour avoir une idée de comment fonctionnent les rotations.

```
/* T1, T2 and T3 are subtrees of the tree rooted with y
   (on left side) or x (on right side)
                y                               x
               / \     Right Rotation          /  \
              x   T3   - - - - - - - >        T1   y
             / \       < - - - - - - -            / \
            T1  T2     Left Rotation            T2  T3 */
```

Figure

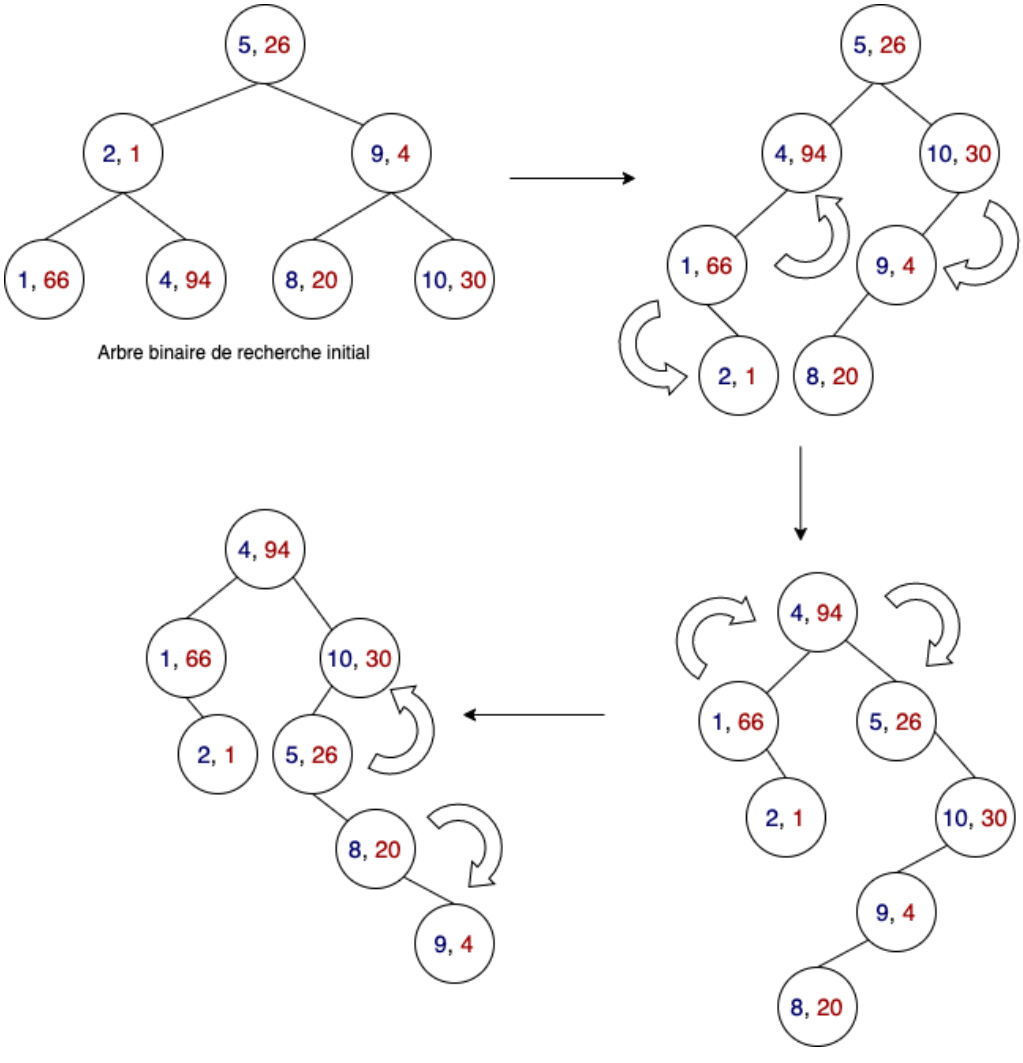
Solution

Code :

```
root = None
treap = Treap()

liste = [5, 2, 1, 4, 9, 8, 10]
nodes = []

# Création de la liste de noeuds + priorités
for i in liste:
    pair = (i, random.randrange(100))
    nodes.append(pair)

print(f"Noeuds avant insertion: {nodes}")

# Construction de la treap
for j in nodes:
    treap.insert(j[0], j[1], root)

print(f"Treap: {treap}")
```

Treap obtenu :

```
[(4,94), [(1,66), None, [(2,1), None, None]], [(10,30), [(5,26), None,
[(8,20), None, [(9,4), None, None]]], None]]
```



5 Chaînes de Markov

Une chaîne de Markov est une suite de variables aléatoires (X_n,$n \in \mathbb{N}$) qui permet de modéliser l'évolution dynamique d'un système aléatoire : X_n représente l'état du système à l'instant n. ("Nous pouvons considérer une chaîne de markov comme une variable aléatoire qui change avec le temps. Le temps discret n est alors un paramètre supplémentaire du système"). La propriété fondamentale des chaînes de Markov, appelée propriété de Markov est que son évolution future ne dépend du passé qu'au travers de sa valeur actuelle : c'est-à-dire que X_{n+1} ne dépend que de X_n et non pas de X_{n-1}, X_{n-2}.
Les applications des chaînes de Markov sont très nombreuses (réseaux, simulation de propagation d'un virus, ingénierie financière, ...). Une chaîne de Markov peut être associée à un graphe orienté où le changement d'état s'effectue suivant une certain probabilité.

Exemple : Durant l'hiver, Pierre peut être dans trois états : en bonne santé (état x_1), enrhumé (état x_2), grippé (état x_3). Son état le jour $n+1$ dépend de son état au jour n et pas des jours précédents :

- S'il est en bonne santé, il le reste le lendemain avec une probabilité égale à 5/6, il s'enrhume avec une probabilité égale à 1/12 et attrape la grippe avec une probabilité égale à 1/12 ;
- S'il est enrhumé il le reste avec une probabilité égale à 1/2, guérit avec une probabilité égale à 1/4 et attrape la grippe avec une probabilité égale à 1/4 ;
- S'il est grippé, il le reste avec probabilité égale à 3/4 et guérit avec une probabilité égale à 1/4.

L'évolution de l'état de santé de Pierre peut être représenté par le graphe de la figure 4.

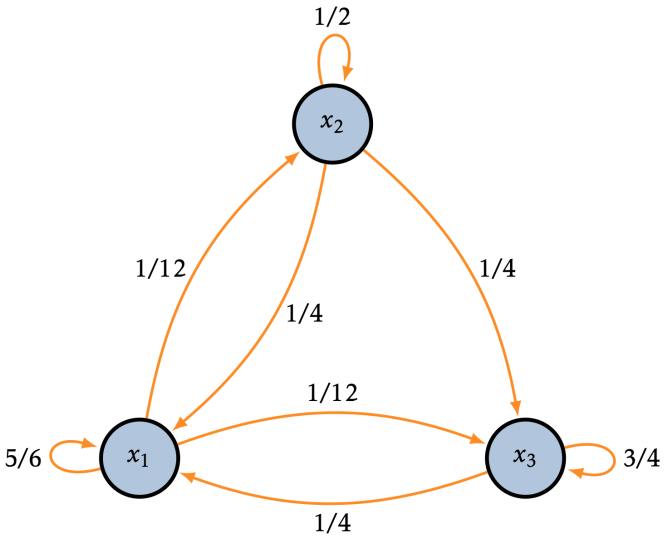
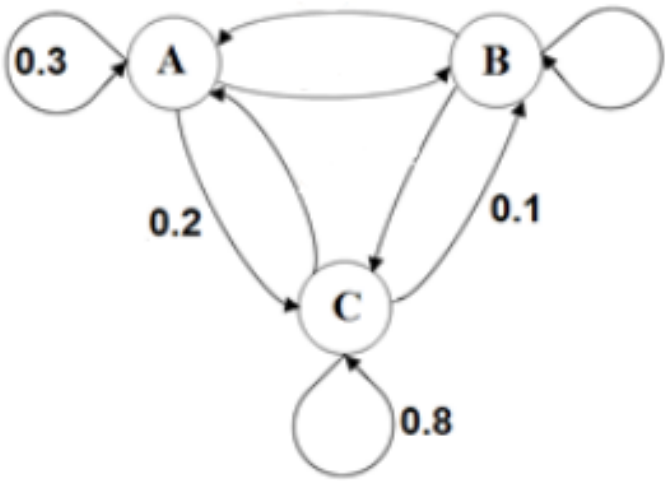


FIGURE 4 – Exemple graph etat de santé de Pierre

La matrice $\mathbf{P}$ dont l'élément à l'indice (i, j) (ligne i, colonne j) p_{ij} est appelée *matrice de transition*. Si on a N états, P est de dimension $N \times N$ (N lignes et N colonnes). Si les chaînes sont homogènes, **$\mathbf{P}$ a deux propriétés importantes** : i, $p_{ij} \geq 0$ et ii, $\sum_i p_{ij} = 1$ (i.e chaque élément de la matrice est supérieur ou égal à 0, et la somme des probabilités à chaque ligne est toujours égale à 1). La matrice de transition associée à l'état de santé de Pierre est donc la suivante :

$$\mathbf{P} = \begin{bmatrix} 5/6 & 1/12 & 1/12 \\ 1/4 & 1/2 & 1/4 \\ 1/4 & 0 & 3/4 \end{bmatrix}$$

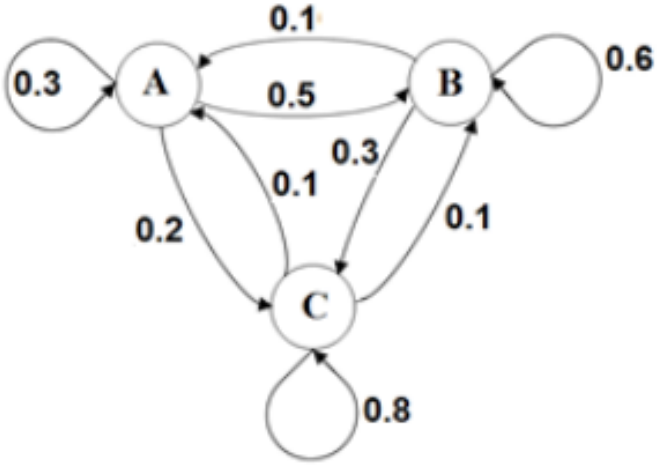
Question 7: (🕓 *5 minutes*) **Complétion de la chaîne de Markov**
On vous donne un diagramme de Markov et sa matrice de transition associée partiellement remplies. Complétez les à l'aide des données déjà inscrites.



$$\mathbf{P} = \begin{bmatrix} . & . & 0.2 \\ 0.1 & . & 0.3 \\ . & . & . \end{bmatrix}$$

Solution

Le diagramme de Markov correspondant est le suivant :



Et sa matrice de transition est :

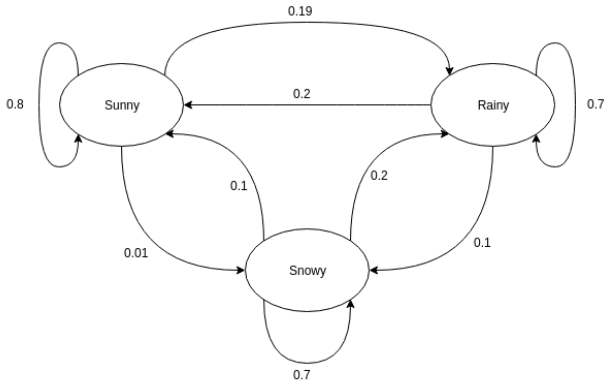
$$\mathbf{P} = \begin{bmatrix} 0.3 & 0.5 & 0.2 \\ 0.1 & 0.6 & 0.3 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}$$

En effet, à partir du graphe on peut constater que la propabilité pour rester dans l'état A sachant que l'état précédent est A est 0.3 donc l'élement à la position [1,1] de la matrice sera égal à 0.3. Ensuite, en utlisant le fait que la somme des probabilités d'une ligne doit être égale à 1 nous trouvons l'élement [1,3] donc p = 1 - 0.3 - 0.2. Il en est de même pour le reste des lignes de la matrice.

Question 8: (*10 minutes*) **Problème - Chaîne de Markov**
Suite à de longs mois d'observation, on a réussi à créer un modèle de prédiction météo en prenant en compte le temps du jour précédent. On sait que lorsqu'il fait beau, les chances qu'il fasse beau le lendemain sont de 80%, celles qu'il pleuve de 19% et celles qu'il neige de 1%. Quand il pleut, les chances qu'il fasse beau le lendemain sont de 20%, celles qu'il pleuve de 70% et celles qu'il neige de 10%. Enfin, quand il neige, les chances qu'il fasse beau le lendemain sont de 10%, celles qu'il pleuve de 20% et celles qu'il neige de 70%. Avec ces informations, dessinez une représentation de Markov sous forme de graphe puis créez la matrice de transition correspondante.

Solution

Le diagramme de Markov correspondant est le suivant :



Et sa matrice de transition est :

$$\mathbf{P} = \begin{bmatrix} 0.8 & 0.19 & 0.01 \\ 0.2 & 0.7 & 0.1 \\ 0.1 & 0.2 & 0.7 \end{bmatrix}$$

Question 9: (*10 minutes*) **Le dé - Chaîne de Markov** Optionnel

Un joueur lance un dé à 6 faces un nombre indéterminé de fois. Chaque chiffre a la même probabilité de sortir à chaque lancer. Tant que le joueur n'a pas obtenu tous les chiffres, il continue à lancer le dé. En assimilant le nombre de chiffres(faces) distincts obtenus aux états d'une chaîne de Markov, représentez sa matrice de transition associée.

Solution

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/6 & 5/6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2/6 & 4/6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3/6 & 3/6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4/6 & 2/6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5/6 & 1/6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Pour la résolution de cet exercice on aura 7 états :

- Etat 0 : avant le lancement du dé, 0 faces obtenus.
- Etat 1 : une face obtenue avec probabilité de 1 si l'état précédent est l'état initial, comme on tombera toujours sur une nouvelle face au premier lancer. Sinon on reste au même état avec une probabilité 1/6 (on tombe sur le même chiffre).
- Etat 2 : deux faces distinctes obtenues, on peut arriver à cet état soit à partir de l'état 1 en tombant sur une nouvelle face avec une probabilité 5/6 ou aussi en restant au même état si on obtient une des deux faces connues avec une probabilité 2/6 .
- ...
- Etat 6 : toutes les faces sont obtenues on peut que retomber sur le même état, donc probabilité égale à 1.