Student: Abdoulaye SAYOUTI SOULEYMANE
Mail: abdoulaye.sayouti-souleymane@alumni.univ-avignon.fr
Group : IA-IL-CLA
ID : uapv2104389

Rapport :

Algorithmes itératifs pour les processus de décision Markovien

1-Programmation dynamique à horizon fini

Dans cette première partie, il est question de programmer, l'algorithme de programmation dynamique avec récursivité inverse afin de déterminer la politique optimale dans un problème particulier. Ce problème est la recherche du chemin le plus coûteux dans un arbre de profondeur T comme illustré sur la figure suivante.

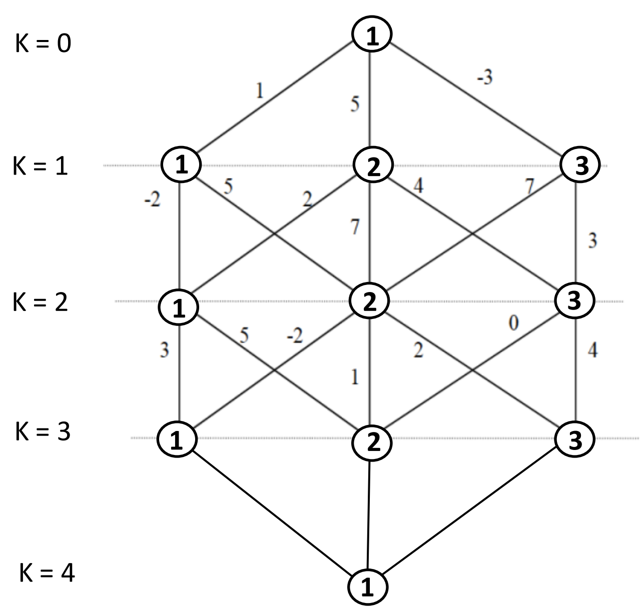


FIGURE 1 – Exemple d'arbre pondéré avec $T = 3$.

Question 1 : Structure de données

- La classe **Graph** est utilisé pour représenter notre graphe. Seule les arrêtes sont enregistrées avec leurs poids. La variable **edges** stock chaque arrête sous le format : *Niceau, Etat -- Niveau,Etat : poid.*

In [1]:
```python
# Classe représentant notre graphe
class Graph:

    def __init__(self):
        self.edges = {} # Seule les arrêtes sont enregistrées
        self.value_functions = {} # Stockage des fonctions de valeur

    # Une fonction qui facilite le stockage des arrêtes dans un dictionnaire
    def to_string(self,k1, s1, k2, s2):
        return str(k1)+','+str(s1)+' -- '+str(k2)+','+str(s2)

    # Stockage du poids de chaque arrête
    def add_edge(self,k1, s1, k2, s2, e):
        self.edges[self.to_string(k1, s1, k2, s2)] = e

    # Affichage du graph
    def print_graph(self):
        print('Niceau,Etat -- Niveau,Etat : poid')
        for k in self.edges.keys():
            print(k,':',self.edges[k])
```

- Les arrêtes sont ajoutés au graphe avec la fonction **add_edge** qui prends 5 paramètres. *(Niveau courant, Etat, Niveau suivant, Etat, poid)*

In [2]:
```python
g = Graph() # Initialisation du graphe

# Ajout des différents arrêtes du graph
# (Niveau, Etat, Niveau, Etat, poid)
g.add_edge(0, 1, 1, 1, 1)
g.add_edge(0, 1, 1, 2, 5)
g.add_edge(0, 1, 1, 3,-3)
g.add_edge(1, 1, 2, 1,-2)
g.add_edge(1, 1, 2, 2, 5)
g.add_edge(1, 2, 2, 1, 2)
g.add_edge(1, 2, 2, 2, 7)
g.add_edge(1, 2, 2, 3, 4)
g.add_edge(1, 3, 2, 2, 7)
g.add_edge(1, 3, 2, 3, 3)
g.add_edge(2, 1, 3, 1, 3)
g.add_edge(2, 1, 3, 2, 5)
g.add_edge(2, 2, 3, 1,-2)
g.add_edge(2, 2, 3, 2, 1)
g.add_edge(2, 2, 3, 3, 2)
g.add_edge(2, 3, 3, 2, 0)
g.add_edge(2, 3, 3, 3, 4)
g.add_edge(3, 1, 4, 1, 0)
g.add_edge(3, 2, 4, 1, 0)
g.add_edge(3, 3, 4, 1, 0)

# Affichage du graphe de la manière suivante: "Niveau,Etat -- Niveau,Etat : poid"
g.print_graph()
```

- La représentation interne du graphe est comme suite :

```
Niceau,Etat -- Niveau,Etat : poid
0,1 -- 1,1 : 1
0,1 -- 1,2 : 5
0,1 -- 1,3 : -3
1,1 -- 2,1 : -2
1,1 -- 2,2 : 5
1,2 -- 2,1 : 2
1,2 -- 2,2 : 7
1,2 -- 2,3 : 4
1,3 -- 2,2 : 7
1,3 -- 2,3 : 3
2,1 -- 3,1 : 3
2,1 -- 3,2 : 5
2,2 -- 3,1 : -2
2,2 -- 3,2 : 1
2,2 -- 3,3 : 2
2,3 -- 3,2 : 0
2,3 -- 3,3 : 4
3,1 -- 4,1 : 0
3,2 -- 4,1 : 0
3,3 -- 4,1 : 0
```

Question 2 : Méthode de récursivité inverse

- Deux fonctions **prev_edges** (renvoie l'ensemble des actions qu'on peut prendre à partir de l'étant courant **s** et du niveau **k)** et **V_star** (déterminer par la méthode de récursivité inverse la fonction de valeur dans chaque état du graphe pour chaque niveau de l'arbre)

```
In [3]: # Fonction permettant de retourner l'ensemble des actions qu'on peut prendre à partir de l'étant courant s et du niveau k
def prev_edges(k,s):
    edges = list(g.edges.keys()) # Récupération de toutes les arrêtes
    items = []
    for e in edges:
        if '- '+str(k)+','+str(s) in e: # Vérification si l'état courant est la destination dans l'arrête
            items.append(e)
    return items

# Fonction permettant de déterminer par la méthode de récursivité inverse la fonction de valeur dans chaque état du graphe
# pour chaque niveau de l'arbre.
def V_star(k,s):
    if k == 0:
        val= 0
    else:
        val = max([g.edges[e] + V_star(int(e[0]),int(e[2])) for e in prev_edges(k,s)]) # Appel récursif de V_star
    g.value_functions['V*(k='+str(k)+',s='+str(s)+')'] = val
    return val
```

- Nous obtenons ce résultat après exécution de la fonction et **V_star** à partir du niveau **k = 4** et de l'état **s = 1.**

```
In [4]: # Calcul des différente fonction de valeur à partir de k = 4, s= 1
V_star(4,1)
for k in g.value_functions.keys():
    print(k,'=',g.value_functions[k])

V*(k=0,s=1) = 0
V*(k=1,s=1) = 1
V*(k=1,s=2) = 5
V*(k=2,s=1) = 7
V*(k=1,s=3) = -3
V*(k=2,s=2) = 12
V*(k=3,s=1) = 10
V*(k=2,s=3) = 9
V*(k=3,s=2) = 13
V*(k=3,s=3) = 14
V*(k=4,s=1) = 14
```

Question 3 : Calcule de la politique optimale et du chemin le plus long depuis la racine (k = 0) jusqu'aux feuilles (k = T).

- La fonction **compute_politique_optimale** détermine la politique optimale à partir du niveau **k** et de l'état **s.**

In [5]:

```python
# Fonction permettant de déterminer la politique optimale

def compute_politique_optimale(k,s):
    politique = [] # Stockage de la politique optimale
    temp1 = g.value_functions['V*(k='+str(k)+',s='+str(s)+')'] # Récupération de la fonction de valeur de la destination k = 4, s
    politique.insert(0,'(k=4,s=1)')
    node =  ''

    # Tant que nous ne somme pas arrivé à l'état d'origine k = 0, s= 1,
    # continuer par déterminer la prochaine action qui possède le maximum comme fonction de valeur
    while k != 0:
        temp2 = prev_edges(k,s)
        temp3 = [g.value_functions['V*(k='+e[0]+',s='+e[2]+')'] for e in temp2]
        val = temp3[0]
        node = temp2[0]
        for index, e in enumerate(temp3[1:], start = 1):
            if e > val:
                node = temp2[index]
                val = e
        k = int(node[0])
        s = int(node[2])
        politique.insert(0,'(k='+str(k)+',s='+str(s)+')')
    return politique
```

- Après exécution de la fonction, nous obtenons le chemin le plus long suivant.

In [6]:

```python
# Affichage du résultat jusqu'à k = 4, s = 1
result = compute_politique_optimale(4,1)
for k in result[:-1]:
    print(k)
    print('    | \n    | \n    v')
print(result[len(result)-1])
print('Le chemin le plus long est : V*(k=4,s=1) =',g.value_functions['V*(k='+str(4)+',s='+str(1)+')'])
```

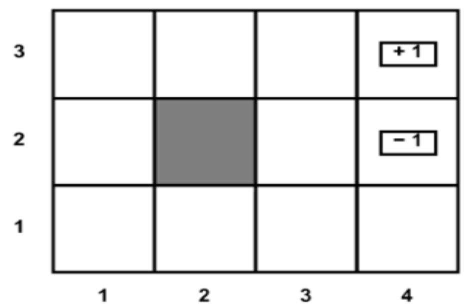
```
(k=0,s=1)
    |
    |
    v
(k=1,s=2)
    |
    |
    v
(k=2,s=2)
    |
    |
    v
(k=3,s=3)
    |
    |
    v
(k=4,s=1)
Le chemin le plus long est : V*(k=4,s=1) = 14
```

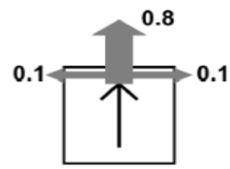
2- Programmation dynamique à horizon infini

L'objectif de cette seconde partie est de programmer, dans le langage de votre choix, les algorithmes itératifs vus en cours pour résoudre les processus de décision Markovien à horizon de temps infini. Ces algorithmes seront testés sur un jeu de déplacements aléatoires d'un robot sur une surface de jeu avec obstacles. Chaque case du plateau correspond à un état a. Les caractéristiques du MDP sont :

- taille du plateau de jeu (3×4) ,
- position de départ du robot [1,1],

- valeur de la récompense dans chaque état s, r = 0,
- probabilités de transition pour chaque action et couple d'états, i.e. P (s′|s, a) : Le robot a une probabilité 0.8 de se déplacer dans la case souhaité, c'est-à-dire auniveau de l'action entrée. Sinon la probabilité est de 0.1 pour les deux direction s perpendiculaire et de 0 pour la direction opposée.
- paramètre d'escompte γ = 0.9.





❖ Initialisation de la grille pour le robot et des différents paramètres d'apprentissage.

- La variable **actions** représente les différentes actions que le robot peut prendre. La variable **directions** permet d'appliquer les actions à un état courant. La variable **forbidden_positions** représente les états dans la grille où nous n'avons pas besoin de calculer la fonction de valeur.

In [1]:
```
# Libraries import
import numpy as np
import math
```

In [2]:
```
# Data initialisation
n = 3
m = 4
gamma = 0.9
start = np.array([2,0])
directions = {'LEFT':np.array([0,-1]),'UP':np.array([-1,0]),'RIGHT':np.array([0,1]),'DOWN':np.array([1,0])}
actions = np.array(['LEFT','RIGHT','UP','DOWN'])
```

In [3]:
```
# Board initialisation
V_board = np.zeros((n,m))
V_board[0,m-1] = 1
V_board[1,m-1] = -1
V_board[1,1] = math.nan
V_board_prev = V_board.copy()
forbidden_positions = [(1,1),(0,3),(1,3)]
print(V_board)
```

```
[[ 0.  0.  0.  1.]
 [ 0. nan  0. -1.]
 [ 0.  0.  0.  0.]]
```

Question 4 : L’algorithme de programmation dynamique par itération de la valeur.

- La fonction **isWall** permet de déterminer si l’état vers le quel on se déplace est un mur.

In [4]:
```
# Function to check if a state is a wall
def isWall(position):
    if position[0] < 0 or position[0] >= n or position[1] < 0 or position[1] >= m or math.isnan(V_board[position[0],position[1]])
        return True
    else:
        return False
```

- La fonction **sum_P_V** permet de calculer la partie encadrée en rouge de la formule.

$$\hat{V}(s) \leftarrow R(s) + \gamma \boxed{\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a)\,\hat{V}(s')}, \quad \forall s \in \mathcal{S}$$

In [5]:
```
# Function that compute the sum  ∑s’∈S p(s’|s, a)Vn(s’)]
def sum_P_V(position, action):
    temp = 0
    next_position = position + directions[action]
    if isWall(next_position):
        temp = temp + V_board_prev[position[0],position[1]]*0.8
    else:
        temp = temp + V_board_prev[next_position[0],next_position[1]]*0.8

    perpendicular_diraction = []
    if action == 'LEFT' or action == 'RIGHT':
        perpendicular_diraction.append('UP')
        perpendicular_diraction.append('DOWN')
    else:
        perpendicular_diraction.append('LEFT')
        perpendicular_diraction.append('RIGHT')


    for a in perpendicular_diraction:
        next_position = position + directions[a]
        if isWall(next_position):
            temp = temp + V_board_prev[position[0],position[1]]*0.1
        else:
            temp = temp + V_board_prev[next_position[0],next_position[1]]*0.1
    return temp
```

- La fonction **value_iteration** calcule la fonction de valeur de chaque état (position) en considérant que le récompense pour tout les autres états est nulle sauf pour l’état de destination (+1) et l’obstacle (-1).

In [6]:
```
# Value iteration function
# We assume that the reward is 0 for all other states
def value_iteration(position):
    return  gamma*max([sum_P_V(position,a) for a in actions])
```

- Ce script correspond au calcule des fonctions de valeur de chaque état après un certain nombre d'itérations.

```
In [7]: #Test of Value Iteration

        nbIteration = 1
        for i in range(nbIteration):
            for k in range(n):
                for l in range(m):
                    if (k,l) not in forbidden_positions:
                        V_board[k,l] = round(value_iteration([k,l]),3)
            V_board_prev = V_board.copy()

        np.set_printoptions(suppress=True)

        print('Number of iteration :', nbIteration)
        print(V_board)
```

  - ➢ Fonction de valeurs pour différentes itérations.

```
Number of iteration : 1
[[ 0.     0.     0.72   1.   ]
 [ 0.       nan  0.    -1.   ]
 [ 0.     0.     0.     0.   ]]
```

```
Number of iteration : 5
[[ 0.585  0.734  0.845  1.    ]
 [ 0.413    nan  0.565 -1.    ]
 [ 0.214  0.306  0.43   0.188]]
```

```
Number of iteration : 100
[[ 0.644  0.744  0.848  1.    ]
 [ 0.565    nan  0.572 -1.    ]
 [ 0.489  0.429  0.475  0.277]]
```

Question 5 : Détermination de la politique optimale par itération de la politique.

- La fonction init_policy_borard permet d'initialiser la grille représentant les politiques.

```
In [9]: # Initialisation of policy board
        def init_policy_board():
            policy_board = []
            for k in range(n):
                policy_board.append([])
                for l in range(m):
                    policy_board[k].append("     ")
            policy_board = np.array(policy_board)
            policy_board[0,3] = 'UP'
            policy_board[1,3] = 'LEFT'
            policy_board_prev = policy_board.copy()

            return policy_board, policy_board_prev
```

- La fonction **policy_iteration** permet de mettre à jour les poliques.

```
In [8]: # Policy iteration function
        def policy_iteration(position):
            sum_P_V_list = [sum_P_V(position,a) for a in actions]
            sum_P_V_max = max(sum_P_V_list)
            policy = actions[np.argmax(sum_P_V_list)]
            return  gamma*sum_P_V_max, policy
```

- Ce script permet de déterminer la politique optimale avec le nombre d'itérations nécéssaires.

```
In [10]: Test of Policy iteration

policy_board, policy_board_prev = init_policy_board()
step = 1
while True:
    for k in range(n):
        for l in range(m):
            if (k,l) not in forbidden_positions:
                result = policy_iteration([k,l])
                V_board[k,l] = round(result[0],3)
                policy_board[k,l]= result[1]
    V_board_prev = V_board.copy()
    step += 1
    if step != 1 and np.array_equal(policy_board,policy_board_prev):
        break
    policy_board_prev = policy_board.copy()
print('Number of iteration : ',step)
print(policy_board)
```

- La politique optimale est obtenue après 7 itérations seulement.

```
Number of iteration :  7
[['RIGHT' 'RIGHT' 'RIGHT' 'UP']
 ['UP' '     ' 'UP' 'LEFT']
 ['UP' 'RIGHT' 'UP' 'LEFT']]
```