AVIGNON UNIVERSITÉ

Rapport TP 1 MDP

Abdou NIANG

29 mars 2023

Master Informatique
Intelligence Artificielle

UE APPRENTISSAGE SUPERVISE
ECUE MODELES STOCHASTIQUES

Responsable
Yezekael Hayel

UFR
SCIENCES
TECHNOLOGIES
SANTÉ

AVIGNON UNIVERSITÉ
CERI

CENTRE
D'**ENSEIGNEMENT**
ET DE **RECHERCHE**
EN **INFORMATIQUE**
ceri.univ-avignon.fr

Sommaire

Titre 1

Sommaire 2

1 Programmation dynamique à horizon fini 3
- 1.1 Structure de données d'arbre pondéré . . . 3
- 1.2 La fonction de valeur dans chaque état du graphe pour chaque niveau de l'arbre . . . 3
- 1.3 Calcule dela politique optimale, le chemin le plus long depuis la racine . . . 5

2 Programmation dynamique à horizon infini 6
- 2.1 Algorithme de programmation dynamique par itération de la valeur . . . 6
- 2.2 Politique optimale par itération de la politique . . . 8

3 Conclusion 9

1 Programmation dynamique à horizon fini

Dans cette partie nous avons choisit d'utiliser le langage de programmation **Python avec la Bibliothèque Numpy**, pour programmer **l'agorithme de programmation dynamique** avec recursité inverse afin de determimer le **chemin le plus long**. Le problème est la recherche du chemin le plus coûteux dans un arbre de profondeur T.

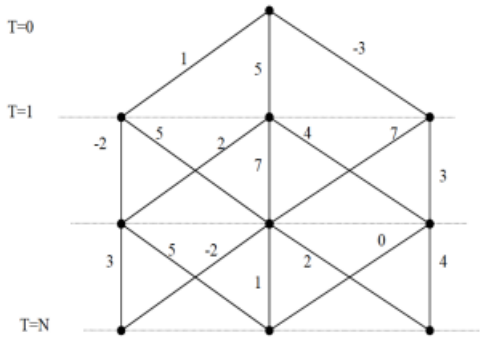

Figure 1. arbre pondérée

1.1 Structure de données d'arbre pondéré

La classe **graph** ci-dessous représente un graphe pondéré avec des fonctions de valeur associées à chaque nœud. Nous l'avons initialisé avec deux dictionnaires vides, un pour les poids des arêtes et l'autre pour les fonctions de valeur. La methode display nous permet d'afficher les relations de notre graph avec les poids.

```
class Graph:

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

    def weight(self,parent, etat_parent,child, etat_child, poid): # le couple
        (child,etat_child) reprente le noeud

        self.poid['(' + str(parent)+','+str(etat_parent)+')' +' <--> '+'('
            +str(child)+','+str(etat_child)+')'] = poid
    def display(self):
        for k in self.poid.keys():
            print(k,':',self.poid[k])
```

Ci-dessous, nous affichons la structure de notre graph, avec la relation entre les états et leurs poids 2.

1.2 La fonction de valeur dans chaque état du graphe pour chaque niveau de l'arbre

Dans cette partie nous avons écrit un programme qui nous permet de déterminer la fonction dans chaque état du graph, par la méthode de la recursivité inverse.
Pour cela, nous avons d'abord définit une fonction **recupe action** quipermet de récupérer toutes les actions possibles à partir d'un **nœud child** avec un **état donné**.

```
g.display()
```

```
(0,0) <--> (1,1) : 1
(0,0) <--> (1,2) : 5
(0,0) <--> (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
```

Figure 2. Display graph avec les poids.

```python
def recupe_action(child,etat_child):
    poid = list(g.poid.keys()) # Récupération de toutes les arrêtes
    items = []
    for p in poid:
        if '-> (' + str(child)+','+str(etat_child)+')' in p :
            items.append(p)
    return items
```

Elle récupère d'abord toutes les arêtes du graphe enregistrées dans le dictionnaire poid avec la méthode keys(). Ensuite, elle parcourt chaque arête et vérifie si elle se termine au nœud child avec l'état etatchild. Si c'est le cas, elle ajoute cette arête à une liste items. Enfin, elle renvoie la liste de toutes les arêtes trouvées.
Ensuite nous définissons une fonction **V etoile** qui nous permet de calculer la fonction de valeur optimale pour chaque état et etatchild donné.

```python
def V_etoile(child, etat_child):
    if child == 0:
        val = 0
    else:
        vals = []
        for p in recupe_action(child, etat_child):
            try:
                vals.append(g.poid[p] + V_etoile(int(p[1]), int(p[3]))) # Appel
                    récursif de V_etoile
            except KeyError: # Si une erreur KeyError est levée, cela signifie que
                l'action n'a pas été trouvée et on l'ignore
                pass
        if vals:
            val = max(vals)
        else:
```

```
            val = float('-inf')
    g.value_functions['V*(k='+str(child)+',s='+str(etat_child)+')'] = val

    return val
```

Enfin nous avons appelé la fonction V etoile en utilisant la boucle for pour afficher les valeurs de toutes les fonctions de valeur optimale.

```
V_etoile(4,1)
for k in g.value_functions.keys():
    print(k,'=',g.value_functions[k])
```

Ci-dessous vous avons les fonction de valeurs optimale pour chaque noeud et état 3.

```
V*(k=0,s=0) = 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
```

Figure 3. Les fonction de valeurs optimale.

1.3 Calcule dela politique optimale, le chemin le plus long depuis la racine

Dans cette partie, nous avons défini une fonction **politique-optimale** qui nous renvoie la politique optimale.
Pour ce faire nous récupèrons toutes les actions possibles à partir de l'état courant (k, s) en appelant la fonction recupe-action. Pour chacune de ces actions, nous récupèrons la valeur de la fonction de valeur optimale associée à l'état de l'action en question à l'aide de la clé **'V*(k='+p[1]+',s='+p[3]+')'** dans le dictionnaire g.valuefunctions. On détermine ensuite l'action qui maximise cette valeur et on la stocke dans la variable noeud.

```
    def V_etoile(child, etat_child):
    if child == 0:
        val = 0
    else:
        vals = []
        for p in recupe_action(child, etat_child):
            try:
                vals.append(g.poid[p] + V_etoile(int(p[1]), int(p[3]))) # Appel
                    récursif de V_etoile
            except KeyError:
                pass
        if vals:
            val = max(vals)
        else:
            val = float('-inf')
    g.value_functions['V*(k='+str(child)+',s='+str(etat_child)+')'] = val
    return val
```

Ainsi nous avons le resultat qui s'affiche ci-dessous 6

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

Figure 4. Politique Optimale

2 Programmation dynamique à horizon infini

Dans cette seconde partie nous avons choisit d'utiliser le langage de programmation **Python** pour programmer les **algorithmes itératifs** pour résoudre les processus de décision Markovien à horizon infini.
Ces algorithmes seront testés sur un jeu de déplacements aléatoires d'un robot sur une surface de jeu avec obstacles.

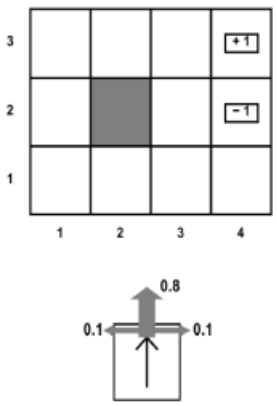

Figure 5. Grille de déplacement du robot

2.1 Algorithme de programmation dynamique par itération de la valeur

Nous avons tout d'abord définir les variables caracteristiques de notre grille : **gamma**, **start**, qui définit la position de départ, **directions** qui est un dictionnaire qui contient les vecteurs de déplacement associés à chaque action possible et **action** qui est tableau qui contient les actions possibles pour l'agent.

```
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'])
```

Nous avons ensuite définit notre plateau appelé **board** qui represente la grille et contient une case contenant **nane** pour la rendre inaccessible. Nous avons aussi définit la variable '**obstacles** qui contient les coordonnées des cases qui sont des obstacles dans la grille.

```
board = np.array([[ 0, 0, 0, 1],
                  [ 0, math.nan, 0, -1],
                  [ 0, 0, 0, 0]])
board_prev = board.copy()
obstacles = [(1,1),(0,3),(1,3)]
print(board)
```

Pour vérifier si la position est valide ou pas, nous avons définit la fonction **verifi-position**.Si la position est en dehors de la grille ou si elle correspond à un état inaccessible (représenté par NaN), la fonction renvoie **True**. Sinon, elle renvoie **False**.

```
def verifi_position(position):
    if position[0] < 0 or position[0] >= n or position[1] < 0 or position[1] >= m or math.isnan(board[position[0],position[1]]):
        return True
    else:
        return False
```

Nous avons définit une fonction **probabilite de transition** qui calcule la probabilité de se déplacer de la position courante à la position suivante en suivant une action donnée. cette fonction utilise la grille board-prev qui est la grille précédente, c'est-à-dire la grille à l'étape t-1.

Notre probabilité de transition est calculée en pondérant la probabilité de se déplacer dans la direction de l'action donnée **(0.8)** et la probabilité de se déplacer dans les deux directions perpendiculaires à l'action **(0.1 chacune)**.

```
def probabilite_de_transition(position, action): # on definit les probabilité de se deplacer de la position courante à la position suivante suivant une action donnée
    sum_pond = 0
    next_position = position + directions[action]
    if verifi_position(next_position):
        sum_pond = sum_pond + board_prev[position[0],position[1]]*0.8
    else:
        sum_pond = sum_pond + board_prev[next_position[0],next_position[1]]*0.8
    direction_perpendiculaire = []
    if action == 'LEFT' or action == 'RIGHT':
        direction_perpendiculaire.append('UP')
        direction_perpendiculaire.append('DOWN')
    else:
        direction_perpendiculaire.append('LEFT')
        direction_perpendiculaire.append('RIGHT')
    for a in direction_perpendiculaire:
        next_position = position + directions[a]
        if verifi_position(next_position):
            sum_pond = sum_pond + board_prev[position[0],position[1]]*0.1
        else:
            sum_pond = sum_pond + board_prev[next_position[0],next_position[1]]*0.1
    return sum_pond
```

Enfin nous definissons la fonction **value iteration** qui calcule la valeur de la case à la position donnée en utilisant la méthode de programmation dynamique de Value Iteration. Nous prendrons en compte toutes les actions possibles (left, right, up, down) et pour chaque

action, la fonction calcule la probabilité de transition vers l'état suivant et la valeur de l'état suivant en utilisant la formule de Bellman qui prend en compte la récompense immédiate et la valeur de l'état suivant pondérée par le facteur de remise gamma

```
def value_iteration(position):
    return gamma*max([probabilite_de_transition(position,a) for a in actions])
```

2.2 Politique optimale par itération de la politique

Nous définissons d'abord une fonction **policy iteration** qui prend en entrée une position sur la grille et calcule sa valeur maximale de la fonction de valeur, ainsi que la politique optimale.
Pour ce faire, la fonction calcule d'abord la liste des probabilités de transition pour chaque action, en appelant la fonction **probabilite de transition(position, a)** pour chaque action a dans actions. Elle détermine ensuite la plus grande probabilité de transition dans cette liste en utilisant la fonction max().

```
def policy_iteration(position):
    probabilite_de_transition_list = [probabilite_de_transition(position,a) for a in
        actions]
    probabilite_de_transition_max = max(probabilite_de_transition_list)
    policy = actions[np.argmax(probabilite_de_transition_list)]
    return gamma*probabilite_de_transition_max, policy

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
```

Nous implémentons enfin la politique d'itération pour résoudre le problème de la grille de Markov.
La fonction init policy board() initialise les tableaux policy board et policy board prev qui contiennent la politique courante et la politique précédente, respectivement. À la fin de la boucle, la politique optimale est affichée sous forme de tableau, ainsi que le nombre d'itérations nécessaires pour atteindre la convergence.

```
policy_board, policy_board_prev = init_policy_board()
it = 1
while True:
    for k in range(n):
        for l in range(m):
            if (k,l) not in obstacles:
                result = policy_iteration([k,l])
                board[k,l] = round(result[0],3)
                policy_board[k,l]= result[1]
    board_prev = board.copy()
    it += 1
    if np.array_equal(policy_board,policy_board_prev):
        break
    policy_board_prev = policy_board.copy()
print(policy_board,'\n')
```

```
16 print('Nombre d iteration est : ',it)
```

Ci-dessous nous avons le résultat montrant les actions à faire sur chaque case notre grille et nombre d'itération pour atteindre la convergence.

```
[['RIGHT' 'RIGHT' 'RIGHT' 'UP']
 ['UP' '      ' 'UP' 'LEFT']
 ['UP' 'LEFT' 'UP' 'LEFT']]

Nombre d iteration est :  3
```

Figure 6. Politique Optimale et nombre d'itération

3 Conclusion

Ce projet nous a permis de mettre en pratique les notions de la programmation dynamique à horizon fini et infinie.