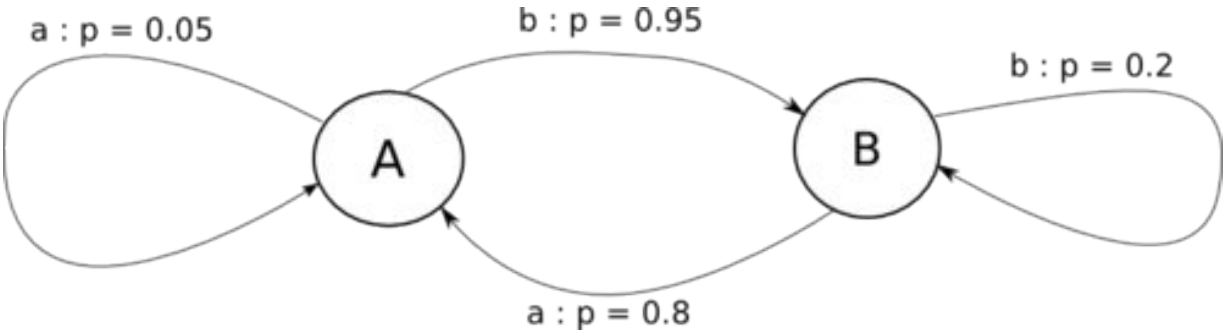
Rapport

Algorithmes itératifs pour les processus de décision Markovien



Alexis SOLA - M1 IA /ILSEN

Modèles stochastiques - Yezekael HAYEL

1. Programmation dynamique à horizon fini

Le but de cette première partie est d'implémenter un algorithme de programmation dynamique à récursivité inverse. Ce dernier à pour objectif de déterminer la politique optimale d'un arbre de profondeur T. La politique optimale est définit comme étant le chemin le plus coûteux de l'arbre.

Nous allons implémenter l'arbre ci-dessous en python afin de tester notre algorithme :

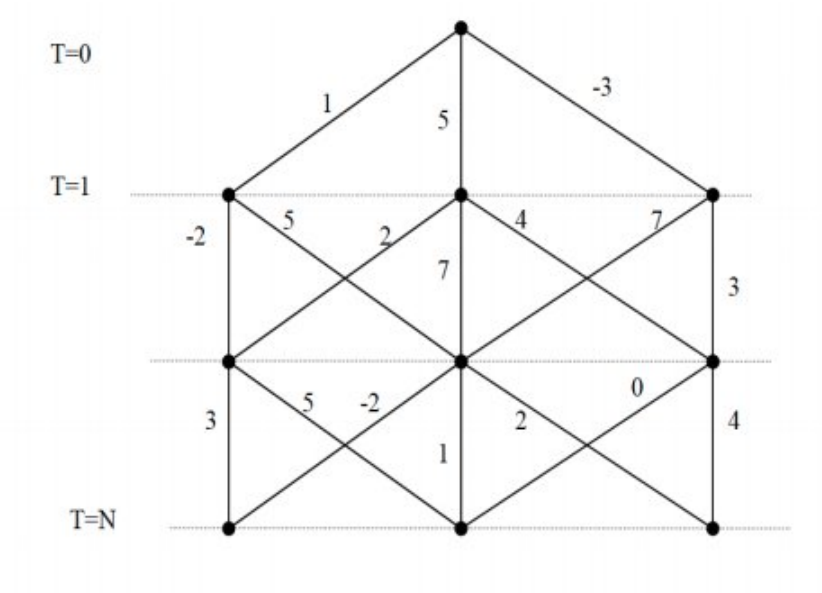


Figure 1 : arbre pondéré avec T = 3

1. Définition d'une structure de donnée

Ici, nous cherchons à définir la structure d'un graphe pondéré avec plusieurs niveau de façon informatique. C'est le graphe vu au dessus que nous allons implémenter. Pour cela nous allons utiliser des tableaux à deux dimensions et des dictionnaires python.

Cette matrice définit les niveaux [0, 1, 2, 3] et les nœuds présents dans chaque niveau.

```
levels = [[1], [2, 3, 4], [5, 6, 7], [8, 9, 10]]
```

Ce dictionnaire permet de savoir à partir d'un nœud, quels nœuds sont atteignables. Par exemple, depuis le nœud 1 il est possible d'aller au nœud 2, 3 et 4.

```
# Which node is linked to the others
nodes = {
    1: [2, 3, 4],
    2: [1, 5, 6],
    3: [1, 5, 6, 7],
    4: [1, 6, 7],
    5: [2, 3, 8, 9],
    6: [2, 3, 4, 8, 9, 10],
    7: [3, 4, 9, 10],
    8: [5, 6],
    9: [5, 6, 7],
    10: [6, 7]
}
```

Enfin, nous avons associé, dans un dictionnaire la aussi, les poids correspondants au passage d'un nœud à un autre.

```
links = {
    "1-->2": 1, "2-->1": 1,
    "1-->3": 5, "3-->1": 5,
    "1-->4": -3, "4-->1": -3,
    "2-->5": -2, "5-->2": -2,
    "2-->6": 5, "6-->2": 5,
    "3-->5": 2, "5-->3": 2,
    "3-->6": 7, "6-->3": 7,
    "3-->7": 4, "7-->3": 4,
    "4-->6": 7, "6-->4": 7,
    "4-->7": 3, "7-->4": 3,
    "5-->8": 3, "8-->5": 3,
    "5-->9": 5, "9-->5": 5,
    "6-->8": -2, "8-->6": -2,
    "6-->9": 1, "9-->6": 1,
    "6-->10": 2, "10-->6": 2,
    "7-->9": 0, "9-->7": 0,
    "7-->10": 4, "10-->7": 4,
}
```

2. Prérequis avant programmation de l'algorithme

Avant toute chose, nous avons préalablement définit des fonctions qui permettront de faciliter la programmation de notre algorithme.

La fonction **IsLinkExist** à pour but de s'assurer qu'un lien existe bien entre deux nœuds. Elle à pour but d'assurer une cohérence dans l'algorithme.

```
# Check if link exist between 2 nodes
def IsLinkExist(begin, end):
    isExist = False
    for val in nodes[begin]:
        if val == end:
            isExist = True
            break
    return isExist
```

La fonction **GetWeight** permet de connaître la valeur du poids entre deux nœuds.

```
# Returns value of weith between 2 nodes if a link exist
def GetWeight(begin, end):
    arrow = "-->"
    if IsLinkExist(begin, end):
        return links[str(begin) + arrow + str(end)]
    else:
        raise Exception("Link does not exist between these two nodes.")
```

ValueLink renvoie tous les poids des les liens pour un nœud. Si l'on rentre le nœud 1 par exemple, la fonction nous renvoie [1, 5, -3]

```
# Get links values of a specific node
def ValueLink(node):
    weight = []
    for val in nodes[node]:
        if node > val:
            weight.append(GetWeight(node, val))
    return weight
```

Enfin, la fonction **GetPreviousNodes** permet de connaître tous les nœuds du niveau inférieurs auxquelles nous avons accès depuis le nœud courant. Elle sera particulièrement utile lorsque nous voudrons parcourir notre arbre de la fin vers les début.

```
# Returns all the previous linked nodes of a specific node
def GetPreviousNodes(node):
    new_tab = []
    for val in nodes[node]:
        if node > val:
            new_tab.append(val)
    return new_tab
```

3. Algorithme de récursivité inverse

Dans cette partie, nous allons voir l'algorithme qui permet de calculer le coût optimal pour un nœud.

Tout d'abord, la fonction **V** prend en paramètre un nœud et renvoie le coût optimal (maximum) permettant d'arriver à ce nœud. Cette fonction va remonter l'arbre, en s'appelant elle même, jusqu'au début.

Voici comment nous avons implémenté cet algorithme :

1. On récupère les nœuds et les poids précédents
2. Si l'on est au minimum au niveau 2 (au niveau 1 les nœuds sont tous associés au nœud 1) :
    i. On parcours tous les poids des nœuds
    ii. On additionne le poids courant au maximum des poids précédents. C'est ici qu'intervient la récursivité de **V** (calcul le maximum des poids précédents)
    iii. On ajoute la nouvelle valeur au tableau calculé et on calcul la valeur maximum
3. Sinon, on est au début, ce n'est pas possible de s'enfoncer plus dans le graphe on renvoie le maximum des poids

Ci-dessous, une copie écran de l'implémentation de la fonction **V** en python.

```python
# Calculate the optimum cost of node
def V(node):
    linked_node = GetPreviousNodes(node)
    w = ValueLink(node)

    if len(linked_node) > 1:
        tab = []
        cpt = 0
        for val in w:
            tab.append(val + V(linked_node[cpt]))
            cpt = cpt + 1
        if len(tab) > 0:
            return max(tab)
    else:
        return max(w)
```

Ensuite, nous avons définit la fonction **ReverseRecur** qui calcul le coût optimal pour un niveau du graphe.

1. On récupère tous les nœuds d'un niveau
2. Pour chaque nœud, on calcule le coût optimal et on l'ajoute au tableau
3. On renvoie le tableau avec les valeur calculées

```python
# Calculate the optimum cost of the graph
def ReverseRecur(level):
    if level >= len(levels) or level <= 0:
        raise Exception("Level doest not exist in this context.")

    nodes = levels[level]
    max_per_nodes = []
    for node in nodes:
        max_per_nodes.append(V(node))
    return max_per_nodes
```

4. Calculer la politique optimal

Après, avoir calculer le coût, nous nous intéressons à trouver le chemin qui nous permettrait d'obtenir ce coût optimal.

Pour cela, nous avons implémenté la fonction **ExtractPolicy** qui renvoie les parcours optimisé pour atteindre le dernier niveau.

1. On parcours tous les niveaux
2. Pour chaque niveau, on calcule les coût de chaque nœud avec la fonction **ReverseRecur**
3. On calcul le coût maximal avec np.argmax()
4. On récupère le nœud associé au coût maximal

```
def ExtractPolicy():
    policy = [1]

    for i in range(len(levels)):
        if i > 0:
            r = ReverseRecur(i)
            argmax = np.argmax(np.array(r))
            policy.append(levels[i][argmax])
    return policy
```

5. Résultat de l'implémentation

Nous allons tester notre programme sur le graphe que nous avons implémenté plus haut. Voici les résultats, coût et police optimale :

```
Coût maximal : 14
Chemin : [1, 3, 6, 10]
```

Le coût maximal trouvé est 14, c'est bien ce que nous avons trouvé lorsque nous avons calculé le coût à la main.

Le chemin optimal est : 1 —> 3 —> 6 —> 10. 1 correspond au nœud du niveau 0, 3 au deuxième nœud du niveau 1, 6 au deuxième nœud du niveau 2 et 10 eu troisième nœud du niveau 3. Si l'on regarde le graphe, le coût total de ce chemin est bien de 14.

Les algorithmes sembles fonctionner correctement.

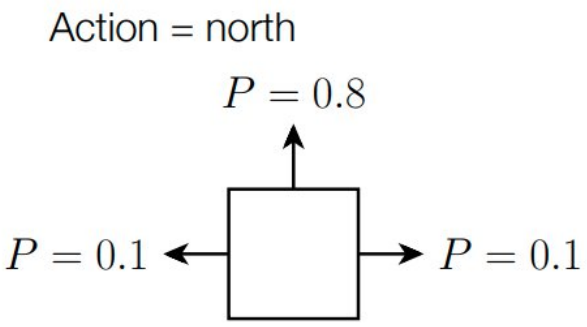
2. Programmation dynamique à horizon infini

Le but de cette partie est d'implémenter les algorithmes itératifs qui permettront de trouver les politique de déplacement optimale d'un robot sur un plateau.

Nous avons choisi de créer un plateau 3x4. L'objectif est que le robot atteigne la case verte qui a une récompense de 1. La case du milieu est considérée comme un mur, le robot ne peut pas se déplacer dedans.

Le robot a une probabilité 0.8 de se déplacer dans la case souhaité, c'est-à-dire au niveau de l'action entrée. Sinon la probabilité est de 0.1 pour les deux autres direction et de 0 pour la direction opposée.

0	0	0	1
0		0	-100
0	0	0	0



1. Structure de données

```
row = 3
col = 4
S = np.zeros((row, col))
S[0][3] = 1
S[1][3] = -100
forbidden = [1, 1]
```

Ici, nous commençons par l'initialisation du plateau S de taille 3x4. Ensuite, nous créer les cases de récompenses (+1 et -100). Enfin, nous ajoutons la case mur ou l'agent ne peut pas aller.

```
V = np.zeros((row, col))

alpha = 0.9
A = ["haut", "bas", "droite", "gauche"]
```

Nous créons une matrice V qui va sauvegarder les valeurs de la politique optimale. Ensuite, nous définissons le facteur de remise et l'ensemble des actions possibles.

```
P = {
    "haut": 0.0,
    "droite": 0.0,
    "gauche": 0.0,
    "bas": 0.0
}

Mirror = {
    "haut": "bas",
    "droite": "gauche",
    "gauche": "droite",
    "bas": "haut"
}

NumberToAction = {
    0: "bas",
    1: "haut",
    2: "gauche",
    3: "droite"
}
```

P associe les probabilités à une action. Pour l'instant les probabilité sont initialisés à 0.

Mirror permet de connaître l'inverse d'une action.

NumberToAction associe un numéro en une action.

2. Prérequis à l'implémentation

Tout d'abord, nous définissons des fonctions qui permettent de savoir si l'agent peut se déplacer dans la case souhaité. 4 fonctions sont créées **MoveUp**, **MoveDown**, **MoveRight** et **MoveLeft**.

On passe un état en paramètre, qui correspond aux coordonnées d'une case (ex : [0, 3]). La fonction vérifie que l'agent ne va pas dans un mur ou dans la case interdite.

```
def MoveUp(state):
    state_copy = copy.deepcopy(state)
    if state_copy[0] == 0:
        return False
    state_copy[0] = state[0] - 1
    if state_copy != forbidden:
        return True
    return False
```

Ensuite, nous compilons les fonctions ci-dessous dans **AuthorizeMove** afin de savoir si l'agent peut se déplacer dans la direction souhaité.

```
def AuthorizeMove(state, direction):
    if direction == "haut":
        return MoveUp(state)
    elif direction == "bas":
        return MoveDown(state)
    elif direction == "droite":
        return MoveRight(state)
    elif direction == "gauche":
        return MoveLeft(state)
    else:
        raise Exception("Direction {0} does not exist.".format(direction))
```

ConvertToAction permet de déplacer l'agent dans un nouvel état si c'est possible, sinon il reste dans son état actuel.

```
def ConvertActionToState(state, direction):
    new_state = copy.deepcopy(state)
    if direction[1] == "haut" and direction[0] == "move":
        new_state[0] = state[0] - 1
    elif direction[1] == "bas" and direction[0] == "move":
        new_state[0] = state[0] + 1
    elif direction[1] == "droite" and direction[0] == "move":
        new_state[1] = state[1] + 1
    elif direction[1] == "gauche" and direction[0] == "move":
        new_state[1] = state[1] - 1
    elif direction[0] == "rester":
        return state
    return new_state
```

WhichDirection renvoie toutes les directions dans lesquelles l'agent peut se déplacer.

```
def WhichDirections(state):
    tab = []
    for direction in A:
        if AuthorizeMove(state, direction):
            tab.append(["move", direction])
        else:
            tab.append(["rester", direction])

    return tab
```

Enfin, la fonction **ChangeP** va nous permettre de changer les probabilités en fonction de l'action effectué par l'agent.

```
def ChangeP(direction):
    P[direction] = 0.8
    for val in A:
        if val != direction:
            P[val] = 0.1
    P[Mirror[direction]] = 0.0
```

3. Algorithme d'itération de la valeur

Nous avons commencé par implémenter l'algorithme permettant de calculer la valeur d'un état. La méthode est appelée **ComputeValue** et fonctionne de la manière suivante :

1. Parcours toutes les actions possibles (haut, bas, gauche, droite).
2. Change la valeur des probabilité en fonction de l'action choisie
3. Parcours toutes les actions que peut faire l'agent
4. Pour chaque action possible, on récupère l'état et on calcul la valeur avec l'équation de Bellman :

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

5. On ajoute la valeur calculée à un tableau. L'objectif est de sauvegarder la valeur v pour chaque action

```
def ComputeValue(state, s, v):
    tab_actions = []
    for action in A:
        tmp = 0
        ChangeP(action)
        for current_action in WhichDirections(state):
            new_state = ConvertActionToState(state, current_action)
            tmp = tmp + P[Mirror[current_action[1]]] * v[new_state[0]][new_state[1]]
        tab_actions.append(s[state[0]][state[1]] + alpha * tmp)

    v_state = max(tab_actions)
    return v_state, tab_actions
```

Après avoir défini la méthode permettant de calculer la valeur v pour un état, nous allons créer l'algorithme d'itération de la valeur. Nommé dans le programme, **ValueIteration** :

1. Parcours tous les états
2. Calcule la valeur v pour chaque état
3. Répète 1 et 2 jusqu'a la convergence, ici nous avons choisi 100 itérations

```
def ValueIteration(nbIter, s, v):
    v_copy = copy.deepcopy(v)
    mat_actions = []
    for val in range(nbIter):
        for i in range(row):
            for j in range(col):
                if [i, j] == forbidden:
                    continue
                value, tab = ComputeValue([i, j], s, v_copy)
                v_copy[i][j] = value
                if val == nbIter - 1:
                    mat_actions.append(tab)
    return v_copy, mat_actions
```

Noter que nous gardons en mémoire dans une matrice le résultat des valeurs de chaque action pour tous les états.

4. Algorithme d'itération de la politique

Précédemment, nous avons calculé les valeurs v pour chaque état de notre système. Maintenant, nous cherchons à extraire la politique optimal pour notre agent. C'est-à-dire, la meilleur action à effectuer pour chaque état afin d'atteindre l'état final.

L'algorithme fonctionne de la manière suivante :

1. Récupérer la matrice (de la dernière itération) contenant les valeurs calculées avec **ValueIteration**
2. Parcourir cette matrice
3. Chercher la valeur max (valeur optimale) pour chaque état et la mapper avec une action. Nous utilisons np.argmax() pour cela

```python
def ExtractPolicy(value_table):
    value_table = InsertValue(value_table)
    policy = []
    tmp = []
    cpt = 1
    i = 0
    for val in value_table:
        if cpt % 5 == 0:
            policy.append(tmp)
            tmp = []
            cpt = 1
        tmp.append(np.argmax(np.array(val)))
        cpt = cpt + 1
        i = i + 1
    policy.append(tmp)
    return policy
```

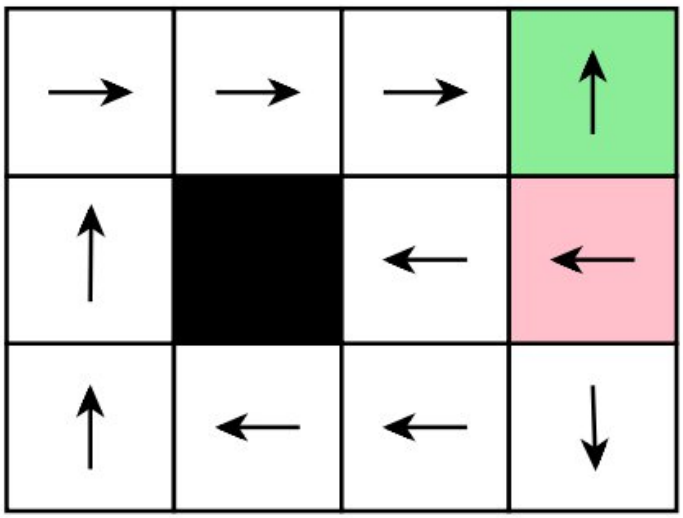
5. Résultat de l'implémentation

Les algorithmes ont déterminé les valeurs de l'équation de Belleman et à partir de ces dernières, ils ont trouvé la politique optimale du déplacement de notre agent. Voici les résultats obtenu :

```
[[  5.4699129    6.31301678   7.18983536   8.66883277]
 [  4.80284863   0.           3.34664644 -96.67286273]
 [  4.16143344   3.65394018   3.22201603   1.5261934 ]]
[['droite' 'droite' 'droite' 'haut']
 ['haut' 'no action' 'gauche' 'gauche']
 ['haut' 'gauche' 'gauche' 'bas']]
```

Les résultats semblent cohérents quand on regarde les déplacements trouvés. Par exemple, si l'agent se trouve en bas à droite la meilleur action à choisir est effectivement 'bas'. Ce choix permet à l'agent d'éviter la probabilité d'aller dans la case du haut étant donné que ses chance d'aller en bas sont 0.8, à droite de 0.1, à gauche de 0.1 et en haut de 0. Les actions choisies semblent suivre cette logique. C'est-à-dire éviter le reward négatif et obtenir le reward positif.

Les actions trouvées sont semblable à l'exemple que nous avons reproduit :



3. Conclusion

Pour conclure, l'implémentation de ces algorithmes pour les processus Markovien m'ont apporté de nouvelles connaissances à plusieurs niveaux :

- Des processus Markoviens
- De la récursivité
- De la programmation dynamique
- De passer d'un modèle théorique à une implémentation concrète des algorithmes vu en cours

Enfin, le dernier exercice m'a permis de comprendre les bases qui composent l'apprentissage par renforcement. Cela a été très intéressant et instructif.