TD4 - Problème de rendu de monnaie

INF TC1 ALGORITHMES ET STRCTURES DE DONNÉES

Auteur :
Alexis DELORME

Enseignant :
Alexandre SAIDI

Le 6 Juin 2020

Sommaire

Introduction

Le but de ce TD est de analyser l'efficacité de trois solutions au problème de rendu de monnaie. Il s'agit de la méthode Gloutonne, une méthode du chemin minimal dans un arbre de recherche et une méthode qui se base dans la Programmation Dynamique. Le problème de rendu de monnaie en forme de question se pose comme suit : quelle est la combinaison de pièces minimale, Q_{Opt}, avec laquelle on peut distribuer un montant M depuis une distributeur avec un ensemble de pièces S de différents valeurs v_i et différents disponibilité d_i ?. Dans ce TD on ne traitera que les pièces en centimes. Afin de classer les méthodes de résolution par ordre de mérite on va tout d'abord définir les exigences que l'on va analyser :

- **Précision.** La solution retenue est-elle satisfaisante ?
- **Temps d'exécution.** La méthode prend-elle trop de temps ?
- **Applicabilité.** La solution est-elle réaliste ?

Si besoin, vous pouvez consulter le répertoire sur github crée pour ce TD :
https://github.com/alexis-delorme/TD4Rendu

Partie I : Algorithme Gloutonne

Question I.1 - Implémentation de l'algorithme

```python
def Monnaie_Gloutonne(S,M):
    Mprim = M
    T=[0]*len(S)
    while Mprim != 0:    #Condition pour ne pas arrêter avant que le montant est rendu
        for idx, valeur in reversed(list(enumerate(S))):    #On parcourt l'inverse de la liste S
            if valeur <= Mprim:                             #Condition pour arrêter à la valeur de la pièce maximale
                T[idx] = Mprim // valeur                    #On garde le quotient de la division euclidienne dans la liste T
                Mprim = Mprim % valeur                      #On soustrait le reste depuis le montant Mprim
    QOptimal = sum(T)
    return f"Montant à rendre : {M}, Liste des pièces utilisées : {T}, Nombre des billets/pièces optimal : {QOptimal}"
```

Avec la méthode Gloutonne on parcourt l'inverse de la liste S pour d'abord traiter les valeurs les plus grandes. Si la valeur divise le montant, $valeur \leq Mprim$, on sauvegarde la quotient de cette division dans la liste T et dont la reste est soustrait du montant pour fournir un nouveau montant Mprim.

Deux tests ont été effectués pour vérifier la bonne fonctionnement de cette fonction : un cas où la fonction marche comme prévue et un cas où la méthode nous donne un optimum local et non pas l'optimum global, voir le script ci-dessous.

```python
print('~~~ Tests pour vérifier la fonctionnement de la méthode Gloutonne ~~~')
S = [1,2,5,10,20,50,100,200,500,1000,2000,5000,10000]
M = 23665
test1=Monnaie_Gloutonne(S,M)
print(test1)

print('\n')
print('Test du cas où la méthode Gloutonne n\'est pas optimale')
S = [1,7,23]
M = 28
test2 = Monnaie_Gloutonne(S,M)
print(test2)
```

Cette partie de code nous donne le output ci-dessous.

```
~~~ Tests pour vérifier la fonctionnement de la méthode Gloutonne ~~~
Montant à rendre : 23665, Liste des pièces utilisées : [0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0,
2],
Nombre des billets/pièces optimal : 9

Test du cas où la méthode Gloutonne n'est pas optimale
Montant à rendre : 28, Liste des pièces utilisées : [5, 0, 1],
Nombre des billets/pièces optimal : 6
```

Comme prévu, la méthode Gloutonne ne marche pas pour le cas où $S = [1, 7, 23]$ et $M = 28$ car $T \neq [0, 4, 0]$ et on a $Q_{opt}(S, M) = 6$ plutôt que $Q_{opt}(S, M) = 4$.

Question I.2 - Problème de Disponibilité

Pour tenir compte de la disponibilité des billets/pièces dans notre distributeur il faut implémenter le pseudo-code modifié ci-dessous

```
Fonction Monnaie_Gloutonne
Entrées : la somme S, M, la disponibilité D
Sorties : le vecteur T, Q et la disponibilité restant D
    M'=M
    Total = 0
    T = le nombre d'occurrences des pièces vi dans la liste S
    Pour chaque pièce vi dans S et disponibilité di dans D:
        Calculer montant total dans la distributeur
        Total += vi * di
    Si Total < M
        Alors rendu impossible car montant trop élevé STOP
    Répéter
        Chercher dans S l'indice i tel que Vi =< M'
        nbBillets = Mprim div valeur
        Si le nombre de billets requis <= nombre de billets disponible:
            Alors Mprim mod valeur
            Ti = nbBillets
            di = di - nbBillets
        Sinon:
            Ti = dispo
            Mprim = Mprim - valeur * dispo
            Di = 0
    Jusqu'à M' = 0 ou la disponibilité de la plus petite valeur D[0] = 0
    $Q = somme de i=1 a i=n de T_i$

    Si la totalité de M n'est pas rendu (Mprim != 0) et D[0] = 0:
        Alors combinaison pas trouvé, return ERROR
    Sinon:
        T, Q et D est la valeur de sortie de l'algorithme
Fin Monnaie_Gloutonne
```

Toute au début on rajoute une condition qui éxige que le montant total dans le distributeur doit être supérieur ou égale à le montant souhaité. On note qu'on a définit une nouvelle condition qui répète l'algorithme jusqu'à ce point où on a utilisé toutes les pièces disponible de la valeur minimale, $D[0]$, pour sortir de l'algorithme puisque il n'y a aucune combinaison qui peut fournir le montant souhaité.

Question I.3 - Implémentation de la modification

Le pseudo-code précisé dans la question I.2 s'écrit en langage python comme le script ci-dessous.

```python
##La fonction de la méthode Gloutonne modifié pour tenir compte de la disponibilité
#En entrée: la liste des valeurs S, le montant M et la liste des disponibilités D
#Conditions initiales : - Le montant M doit être un nombre entier
#                       - La liste D doit être de la même taille que la liste S
def Monnaie_Gloutonne_Modifie(S,M,D):
    Mprim = M
    T=[0]*len(S)
    Total = 0
    for idx, valeur in list(enumerate(S)):  #On parcourt l'inverse de la liste S
        Total += valeur*D[idx]
    if Total < M:                           #Condition pour arrêter si le montant est trop élevé
        return f"Insuffisament des billets/pièces... Montant total = {Total}"

    while Mprim != 0 and D[0] > 0:  #On rajoute la condition D[0]>0 pour arrêter si il n'y a plus de pièces pour la valeur minimale
        for idx, valeur in reversed(list(enumerate(S))):
                dispo = D[idx]
                nbBillets = Mprim // valeur
                if nbBillets <= dispo:          #Le cas où on supprime que le nombre de pièces utilisé
                    Mprim = Mprim % valeur
                    T[idx] = nbBillets          #On garde combien de fois de la valeur a été utilisé dans la liste T
                    D[idx] = dispo - nbBillets
                else:                           #Sinon, on supprime le dispo entièrement dans la liste D
                    T[idx] = dispo
                    Mprim = Mprim - valeur*dispo
                    D[idx] = 0
    QOptimal = sum(T)
    if Mprim != 0 and D[0] == 0:    #Lorsque D[0]=0 on sorte du loop while et on vérifie si tout le montant a été rendu
        return f"Rendu impossible, aucune combinaison des billets trouvé pour ce montant"
    return f"Liste des pièces : {T}, Nombre des billets/pièces optimal : {QOptimal}, Disponibilité de pièces : {D}"
```

Avec les nouvelles conditions on n'a plus besoin de la condition $v_i \leq Mprim$ car si cela est le cas la division euclidienne $Mprim//v_i = 0$ et le nouveau montant devient le même : $Mprim\%valeur = Mprim$. Alors on répète l'algorithme en essayant avec la valeur v_{idx-1} et avec la même valeur de $Mprim$ et les valeurs de $T[idx]$ et $D[idx]$ ne changent pas.

Les trois tests effectués sont décrits dans le script ci-dessous

```python
print('\n')
print('~~~ Tests pour vérifier la fonctionnement de la méthode Gloutonne modifié ~~~')
print('Test d\'un cas où la disponibilité limite le choix qu\'on pourrait faire')
S = [4,7,23]
D = [3,5,0]
M = 28
test3 = Monnaie_Gloutonne_Modifie(S,M,D)
print(test3)

print('\n')
print('Test d\'un cas où le montant est trop élevé')
S = [1,2,3,4]
D = [0,1,1,1]
M = 12
test4 = Monnaie_Gloutonne_Modifie(S,M,D)
print(test4)


print('\n')
print('Test d\'un cas où il n\'y a aucune combinaison possible pour le montant M')
S = [100,300,450,646,1500]
D = [0,10,6,5,3]
M = 12001
test5 = Monnaie_Gloutonne_Modifie(S,M,D)
print(test5)
```

Ce qui fournit le output :

```
~~~ Tests pour vérifier la fonctionnement de la méthode Gloutonne modifié ~~~
Test d'un cas où la disponibilité limite le choix qu'on pourrait faire
Liste des pièces : [0, 4, 0], Nombre des billets/pièces optimal : 4, Disponibilité de pièces
: [3, 1, 0]

Test d'un cas où le montant est trop élevé
Insuffisament des billets/pièces... Montant total = 9

Test d'un cas où il n'y a aucune combinaison possible pour le montant M
Rendu impossible, aucune combinaison des billets trouvé pour ce montant
```

Partie II : Chemin minimal dans un arbre

Question II.1 - Construction de l'arbre

Avec le pseudo-code donné dans l'énoncé le script ci-dessous a été crée.

```python
## La fonction de la méthode du chemin minimale d'arbre
#En entrée : la liste des valeurs S, le montant M
#Conditions initiales : - Pour avoir une solution pour chaque montant M il faut que S commence à 1.
#                       - Le montant M doit être un nombre entier
def Monnaie_Graphe(S,M):
    FileF = []          #Création de la file d'attente
    FileF.append(M)
    Noeuds = [M]
    ArbreA = [[M,[]]]       #Création de l'arbre
    while FileF != []:      #Condition pour arrêter si la file d'attente est vide
        Parent = FileF[0]
        for valeur in S:
            noeud = Parent - valeur     #Chaque noeud représent un montant Mprim < M
            if valeur < Parent:
                if noeud not in Noeuds:
                    Noeuds.append(noeud)
                    ArbreA.append([noeud, [Parent]])
                    FileF.append(noeud)
                else:
                    index = [y[0] for y in ArbreA].index(noeud)
                    ArbreA[index][1].append(Parent)
            if noeud == 0:                      #Si on a trouvé un chemin qui vient d'arriver au montant Mprim=0 on arrête
                ArbreA.append([0,[Parent]])
                return ArbreA, Noeuds
        FileF.pop(0)                        #"Réinitialisation" de la file d'attente
    return (ArbreA, Noeuds)
```

La condition "Jusqu'à $Mprim - v_i = 0$" ne permet pas la création de toutes combinaisons possibles mais plutôt toutes les combinaisons jusqu'au noeud qui vaut 0. Si on enlève cette condition le code marche bien sauf que le code devient beaucoup plus lourd à exécuter, même avec cette condition mise en place la méthode ne peut traiter que des listes S assez courts et montants M faibles.

Ensuite, en utilisant graphviz nous pouvons visualiser l'arbre avec ses noeuds, parents et liens avec la partie de code ci-dessous. La résultat de cette fonction pour deux arbres différents est illustré sur les **Figures 1** et **2**. Ces figures illustrent clairement que l'arbre devient très grand et compliqué très vite.

```python
##Fonction pour dessiner l'arbre
#En entrée : Le tuple avec l'arbre et les noeuds produit avec la fonction Monnaie_Graphe
def Graph_Arbre(arbre_noeuds):
    arbre = arbre_noeuds[0]
    for liens in arbre:     #On parcourt chaque lien dans l'arbre
        Parents = liens[1]      #On identifie les noeuds et les parents
        noeud = str(liens[0])
        for parent in Parents:
            graph.edge(str(parent), noeud, label=str(parent - int(noeud)))      #Création avec graphviz
    print(graph.source)
    graph.render('Arbre.gv', view=True)     #Nom de la sauvegarde, view=true affiche l'arbre
```

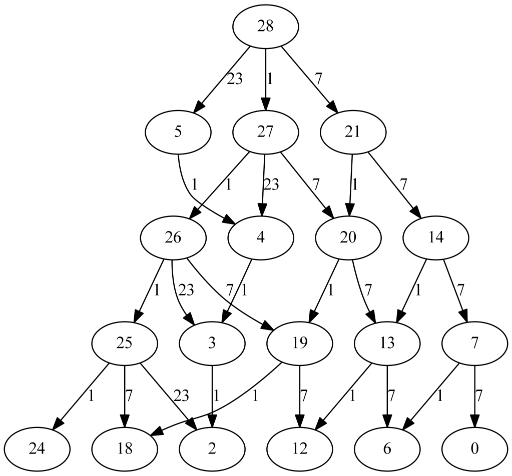

Figure 1 – Arbre crée avec $M = 28$ et $S = [1, 7, 23]$

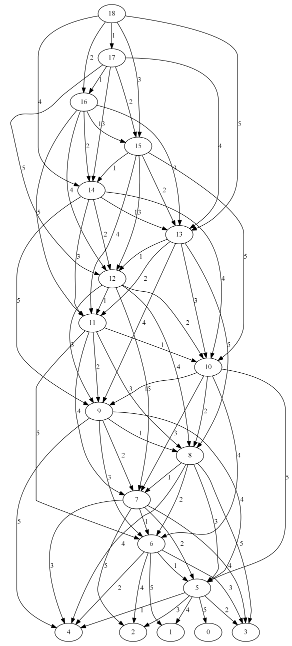

Figure 2 – Arbre crée avec $M = 18$ et $S = [1, 2, 3, 4, 5]$

Question II.2 - Recherche du chemin le plus court

Afin de trouver le chemin le plus court dans l'arbre on crée une nouvelle fonction Q_Optimal qui prend en entrée la sortie de la fonction Monnaie_graphe, la liste S et le montant M.

```
##Fonction pour calculer la combinaison optimale
#En entrée : Le tuple avec l'arbre et les noeuds produit avec la fonction Monnaie_Graphe, S et M
#Conditions initiales : même que pour Monnaie_Graphe
def Q_Optimal(arbre_noeuds, S, M):
    arbre = arbre_noeuds[0]
    temp=0
    T = [0]*len(S)
    while M != 0:            #Parcours de l'arbre
        temp1 = temp
        index = [y[0] for y in arbre].index(temp)
        if index == 0:
            return f"Liste des pièces utilisées optimale : {T}"
        temp = arbre[index][1][0]
        M -= temp
        valeur = S.index(temp - temp1)
        T[valeur] += 1
    return f"Liste des pièces utilisées optimale : {T}"
```

Les tests effectués pour trouver le chemin le plus court sont décrits dans le code ci-dessous, y compris un test pour trouver le temps écoulé lors d'exécution de la fonction.

```
print('\n')
print('~~~ Tests pour vérifier la fonctionnement de la méthode de l\'arbre ~~~')
S = [1,10,200,300,4000]
M = 4654
start = time.time()
test6 = Monnaie_Graphe(S,M)
Qopt = Q_Optimal(test6,S,M)
end = time.time()
print('Temps écoulé = ', end - start, 'seconds')
print(Qopt)

print('\n')
print('Test du cas où la méthode Gloutonne a échouée')
S = [1,7,23]
M = 28
test7 = Monnaie_Graphe(S,M)
print(Q_Optimal(test7,S,M))
```

Le output de ces tests :

```
~~~ Tests pour vérifier la fonctionnement de la méthode de l'arbre ~~~
Temps écoulé =  0.3152010440826416 seconds
Liste des pièces utilisées optimale : [4, 5, 0, 2, 1]


Test du cas où la méthode Gloutonne a échouée
Liste des pièces utilisées optimale : [0, 4, 0]
```

Le temps écoulé semble un peu variable lors de chaque exécution du code mais en l'éxécutant plusieurs fois on peut en déduire un temps moyen d'environ 0.3 **seconds** pour la liste $S = [1, 10, 200, 300, 4000]$ et le montant $M = 4654$. En outre, la fonction Monnaie_Graphe réussi a traître les cas différents et notamment celui que la méthode Gloutonne a échoué.

Partie III : Algorithme de Programmation Dynamique

Question III.1 - Recherche du nombre minimal de pièces

Guidé par le pseudo-code dans l'énoncé le code ci-dessous a été crée. On note que le nombre de combinaisons optimal se trouve en bas à droite de la matrice.

```python
##Fonction de la méthode de Programmation Dynamique
#En entrée : la liste des valeurs S, le montant M
#Conditions initiales : - Pour avoir une solution pour chaque montant M il faut que S commence à 1.
#                       - Le montant M doit être un nombre entier
def Monnaie_dynamique(S,M):
    Szero=S
    Szero.insert(0,0)
    w, h = M+1, len(Szero)
    mat = [[0 for x in range(w)] for y in range(h)]        #Création de la matrice mat[S][M]
    for i in range(len(Szero)):
        for m in range(M+1):
            temp = []
            if i == 0:
                mat[i][m] = float('inf')
            elif m == 0:
                mat[i][m] = 0
            else:
                if m - Szero[i] >= 0:
                    temp.append(1 + mat[i][m - Szero[i]])
                else:
                    temp.append(float('inf'))
                if i >= 1:
                    temp.append(mat[i-1][m])
                else:
                    temp.append(float('inf'))
                mat[i][m] = min(temp)
    return mat[i][m]          #La combinaison minimale pour le montant M se trouve en bas à droite dans la matrice
```

Les tests effectués sont décrits par le code ci-dessous.

```python
print('\n')
print('~~~ Tests pour vérifier la fonctionnement de la méthode Programmation Dynamique ~~~')
S = [1,10,200,300,4000]
M = 4654
start1 = time.time()
test8 = Monnaie_dynamique(S,M)
end1 = time.time()
print('Temps écoulé = ', end1 - start1, 'seconds')
print('Combinaison optimale (QOpt) :', test8)

print('\n')
print('Test d\'un cas avec un nombre des combinaisons important')
S = [1,2,5,10,20,50,100,200,500,1000,2000,5000,10000]
M = 23665
start2 = time.time()
test9 = Monnaie_dynamique(S,M)
end2 = time.time()
print('Temps écoulé = ', end2 - start2, 'seconds')
print('Combinaison optimale (QOpt) :', test9)
```

Le output de ces tests :

```
~~~ Tests pour vérifier la fonctionnement de la méthode Programmation Dynamique ~~~
Temps écoulé =  0.04787015914916992 seconds
Combinaison optimale (QOpt) : 12

Test d'un cas avec un nombre des combinaisons important
Temps écoulé =  0.4288508892059326 seconds
Combinaison optimale (QOpt) : 9
```

Avec cette fonction pour la même liste $S = [1, 10, 200, 300, 4000]$ et le montant $M = 4654$ on trouve un temps moyen d'exécution de 0.045 **seconds**, alors à peu près dix fois plus vite que la fonction Monnaie_graphe (+ la fonction q_Optimal). On voit également que cette méthode fonctionne bien même dans un cas avec un nombre de combinaisons très important (presque impossible à exécuter avec la méthode de l'arbre). Par contre avec cette méthode on trouve seulement le *nombre de combinaisons minimal* et non pas les pièces utilisées mais on va voir que cela n'importe peu sur le temps écoulé.

Question III.2 - Solution pour trouver les pièces utilisées

Le code ci-dessous a été utilisé pour trouver les pièces utilisés, où les seules lignes rajoutées par rapport au code pour la question III.1 sont 145 et 165-171 (et que on utilise S et non pas Szero pour parcourir la matrice). Les lignes rajoutées permet le parcours de la matrice pour trouver la combinaison minimale pour un montant donné et des pièces disponible.

```python
140 ##Fonction de la méthode de Programmation Dynamique modifiée
141 #En entrée : la liste des valeurs S, le montant M
142 #Conditions initiales : - Pour avoir une solution pour chaque montant M il faut que S commence à 1.
143 #                       - Le montant M doit être un nombre entier
144 def Monnaie_dynamique_modifie(S,M):
145     T = [0]*len(S)
146     w, h = M+1, len(S)+1
147     mat = [[0 for x in range(w)] for y in range(h)]
148     for i in range(len(S)+1):
149         for m in range(M+1):
150             temp = []
151             if i == 0:
152                 mat[i][m] = float('inf')
153             elif m == 0:
154                 mat[i][m] = 0
155             else:
156                 if m - S[i-1] >= 0:
157                     temp.append(1 + mat[i][m - S[i-1]])
158                 else:
159                     temp.append(float('inf'))
160                 if i >= 1:
161                     temp.append(mat[i-1][m])
162                 else:
163                     temp.append(float('inf'))
164                 mat[i][m] = min(temp)
165     matmodi = mat[1:]           #Partie rajoutée
166     while M != 0:               #Parcours de la matrice matmodi pour trouver le chemin minimale
167         k = len(S) - 1
168         while k > 0 and matmodi[k][M] == matmodi[k-1][M]:
169             k -= 1
170         T[k] += 1
171         M -= S[k]
172     return [mat, mat[i][m], T]
```

Les tests effectués sont décrits dans le code ci-dessous :

```python
print('\n')
print('~~~ Tests pour vérifier la fonctionnement de la méthode Programmation Dynamique modifié ~~~')
S = [1,10,200,300,4000]
M = 4654
start3 = time.time()
test10 = Monnaie_dynamique_modifie(S,M)
end3 = time.time()
print('Temps écoulé = ', end3 - start3, 'seconds')
#print('Matrice finale : ',test10[0])
print('Combinaison optimale (QOpt) :', test10[1])
print('Liste des pièces utilisées :', test10[2])
```

Avec le output de ces tests :

```
~~~ Tests pour vérifier la fonctionnement de la méthode Programmation Dynamique modifié ~~~
Temps écoulé =  0.04089188575744629 seconds
Combinaison optimale (QOpt) : 12
Liste des pièces utilisées : [4, 5, 0, 2, 1]
```

Le temps écoulé en moyen pour cette fonction étant à peu près égale au temps moyen de la fonction Monnaie_Dynamique on en déduit que le temps écoulé pour parcourir la matrice est presque nul et que c'est plutôt la chargement de la matrice qui prend la plupart du temps.

Question III.3 - Complexité en espace/temps

La complexité en espace/temps de la solution Monnaie_dynamique_modifie est égale au nombre d'éléments dans la matrice mat[S][M]. Le code pour trouver cette valeur est assez simple, il suffit de compter les nombres des lignes et colonnes de notre matrice.

```python
##Fonction pour calculer la complexité en espace/temps
#En entrée : La matrice mat fournie par la fonction Monnaie_dynamique_modifie
def Complexite(mat):
    matList = list((j for i in mat for j in i))
    return f"La complexité en espace/temps = {len(matList)}"

```

Ce sera intéressant de faire une étude sur la relation entre cette complexité en espace/temps et le temps écoulé de l'exécution afin de voir quelle est la nature de cela. Mon hypothèse est que elle est linéaire d'après quelques petits tests que j'ai fait.

Partie Bonus

Les tests sur eclope ont été effectués.

Conclusion

En somme, chacune des trois méthodes présente ses propres avantages et inconvénients et il faut un cahier de charge pour préciser les exigences indiquées dans l'Introduction sur lesquelle il faut s'appuyer. Par exemple, on ne peut pas s'assurer que la méthode Gloutonne nous rend le nombre de pièces que l'on a souhaité mais cette méthode nous donne une résultat très vite par rapport aux autres méthodes face à des montants et listes de valeurs très grands. En revanche, si on ne permet aucune faute de précision il faut choisir une de deux autres méthodes. Ensuite la choix entre la méthode du chemin minimal dans un arbre de recherche et la méthode de la Programmation Dynamique est claire : cette dernière étant presque dix fois plus rapide que celle d'avant. La programmation dynamique peut être appliquée dans une vraie distributeur car une fois la matrice remplie il n'en reste que du parcours de la matrice, et on vient de découvrir que cela ne prend presque pas de temps. Cependant, notre script ne prend pas en compte les disponibilités de chaque pièce et alors il faut améliorer la méthode dans cette aspect là.