ÉCOLE CENTRALE LYON

Rapport d'informatique tc1 td 5

Groupe A2b

Rendu de Monnaie

Auteur :
M Valentin VERDON

Encadrant :
M. Stéphane DERODE

Version du
19 mai 2022

Table des matières

1 Introduction

Le problème de rendu de monnaie permet d'aborder la notion de programmation dynamique. En effet, si nous souhaitons rendre la monnaie en optimisant le nombre de pièces ou bien le poids du rendu (problème du sac à dos) plusieurs implémentation (Algorithme Glouton, parcours d'arbre, programmation dynamique) plus ou moins performantes existent.

Dans ce rapport, nous ne traiterons que l'implémentations par Programmation dynamique.

2 Rendu avec le moins de pièce

Dans cette partie, le programme doit permettre de rendre, pour un système de monnaie donné et une somme donnée, la solution avec le moins de pièce à rendre.

2.1 Principe de l'algorithme

Ici, on va utiliser le principe de mémoîsations. C'est-à-dire que l'on va chercher le résultat optimal pour tous les rendus inférieur au rendu souhaité. De plus, on fera cette simulation en ajoutant les pièces au fur et à mesure dans le système de monnaie. Ainsi, à chaque étape, sous l'ajout d'une pièce dans le système de monnaie ne permet pas d'avoir une solution plus optimale et on reprend donc la solution du système de monnaie précédent. Sinon, on prend la solution avec un rendu égale au rendu souhaité moins la valeur de la nouvelle pièce.

Voici un exemple du tableau généré par l'algorithme pour un rendu de 7 et le système de monnaie {1,3,4}.

	0	1	2	3	4	5	6	7
∅	0	∞	∞	∞	∞	∞	∞	∞
1c	0	1	2	3	4	5	6	7
1c, 3c	0	1	2	1	2	3	2	3
1c, 3c, 4c	0	1	2	1	1	2	2	2

+1

FIGURE 1 – Exemple tableau rendu de monnaie

2.2 Implémentations de l'algorithme

Dans un premier temps, on implémente le programme présenté précédemment avec S la liste des valeurs des pièces du système de monnaie et M le rendu de monnaie à effectuer.

```
def Monnaie(S,M):
    mat=[[0 for i in range(M+1)] for i in range(len(S)+1)]
    for i in range(len(S)+1):
        for m in range(M+1):
            if m==0:
                mat[i][m]=0
            elif i==0:
                mat[i][m]=float('inf')
            else:
                aux1,aux2=float('inf') ,float('inf')
                if m-S[i-1]>=0:
                    aux1=1+mat[i][m-S[i-1]]
                if i>=1:
                    aux2=mat[i-1][m]
                mat[i][m]=min(aux1,aux2)
    return mat[len(S)][M]
```

FIGURE 2 – Algorithme minimisation rendue pièce

Si on test le programme avec le système de monnaie précédent nous avons bien en sortie 2 pièces.

Pour améliorer ce programme, nous pouvons choisir de mettre en sortie les pièces utilisées ainsi que de gérer un stock de pièces.

2.3 Amélioration de l'algorithme

Afin de savoir quelle pièce a été utilisée pour atteindre cette case, nous utilisons un espace de stockage supplémentaire qui mémorise les pièces utilisées au fur et à mesure. (Remarque : nous aurions pu utiliser un algorithme de backtracking qui à l'avantage de ne pas utiliser d'espace de stockage supplémentaire)

Pour ce qui est du système de stock de pièces, nous ajoutons une condition pour utiliser des pièces : il faut qu'il y en ait dans le stock. Si c'est le cas, on enlève une pièce au stock pour cette configuration. Ainsi, chaque case possède 3 informations : le nombre de pièces utilisé pour faire le rendu, la liste des pièces ainsi que le stock restant. (Remarque : pour le stock des pièces nous aurions pu ne garder que les pièces utilisées et en déduire à l'aide d'une fonction annexe retrouver le stock de pièces)

Voici la seconde version du programme avec D la liste des pièces disponibles :

```python
def Monnaie_D(S,M,D):
    mat=[[[0,[],D] for i in range(M+1)] for i in range(len(S)+1)]
    for i in range(len(S)+1):
        for m in range(M+1):
            if m==0:
                mat[i][m]=[0,[],D]
            elif i==0:
                mat[i][m]=[float('inf'),[],D]
            else:
                aux1,aux2=[float('inf'),[],D] ,[float('inf'),[],D]
                if m-S[i-1]>=0 and mat[i][m-S[i-1]][2][i-1]>0:
                    aux1=[1+mat[i][m-S[i-1]][0],mat[i][m-S[i-1]][1].copy(),mat[i][m-S[i-1]][2].copy()]
                    aux1[1].append(S[i-1])
                    aux1[2][i-1]-=1
                if i>=1:
                    aux2=mat[i-1][m]
                if aux1[0]>aux2[0]:
                    mat[i][m]=aux2
                else:
                    mat[i][m]=aux1
    if mat[len(S)][M][0]==float('inf'):
        return "On ne peut pas rendre la monnaie"
    return mat[len(S)][M]
```

FIGURE 3 – Algorithme minimisation rendue pièce avec stock

On pourra remarquer que l'on utilise la méthode *.copy* pour copier les listes car si on faisait simplement une égalité de liste alors nous changerions la valeur de la liste initiale en même temps que nous changerions la valeur de la liste auxiliaire.

Ainsi, si pour le système précédent nous implémentons le stock suivant : { 10,0,2}, nous avons en sortit :

```
[4, [1, 1, 1, 4], [7, 0, 1]]
```

FIGURE 4 – Sortie de la fonction

C'est effectivement le choix minimum de pièce car nous n'avons pas de pièce de 3 pour rendre la monnaie, notre algorithme fonctionne !

3 Problème de minimisation du poids (backpack problem)

3.1 Présentation du problème

Nous allons maintenant prendre en compte le poids des pièces et choisir de minimiser le rendu de pièce, non pas en fonction du nombre de pièces, mais

plutôt en fonction du poids total. Pour cela nous allons commencer par prendre un exemple :

Nous souhaitons rendre *M*=7 avec le système suivant :

S (valeur de chaque pièce)	P (poids de chaque pièce)
1c	2,30g
2c	3,06g
5c	3,92g
10c	4,10g
20c	5,74g
50c	7,80g
100 (1€)	7,50g
200 (2€)	8,50g
500 (5€)	0,6g
1000	0,7g
2000	0,8g
5000	0,9g
10000	1g

FIGURE 5 – Système n°1

Ici, la solution sera une pièce de 5 et une pièce de 2 de poids total 6,98g qui est également la solution qui minimise le nombre de pièces.

Si maintenant nous choisissons de rendre *M*=6 avec le système suivant :

S (valeur de chaque pièce)	P (poids de chaque pièce)
1c	10g
3c	27g
4c	32g
7c	55g

FIGURE 6 – Système n°2

Ici la solution optimale en poids est une pièce de 4 et deux pièces de 1 pour un poids totalde 52g alors que la solution optimale en pièce est de deux pièces de 3 pour un poids total de 54g.

3.2 Principe de l'algorithme

Pour résoudre ce problème, nous allons utiliser la programmation dynamique. En reprenant la structure du programme de la première partie, nous allons choisir de prendre la solution avec la nouvelle pièce ou non en se basant sur le poids total et non sur le nombre de pièces utilisé.

3.3 Implémentation

Nous obtenons le code suivant :

```
def Monnaie_Poids(S,P,M):
    mat=[[[0,0] for i in range(M+1)] for i in range(len(S)+1)] #on construit le tableau avec (reste,poids)
    for i in range(len(P)+1):
        for m in range(M+1):
            if m==0:
                mat[i][m]=[0,0]
            elif i==0:
                mat[i][m]=[float('inf'),float('inf')]
            else:
                aux1,aux2=[float('inf'),float('inf')] ,[float('inf'),float('inf')]
                if m-S[i-1]>=0:
                    aux1=[1+mat[i][m-S[i-1]][0],mat[i][m-S[i-1]][1]+P[i-1]]
                if i>=1:
                    aux2=mat[i-1][m]
                if aux1[1]>aux2[1]:
                    mat[i][m]=aux2
                else:
                    mat[i][m]=aux1
    return mat[len(S)][M]
```

FIGURE 7 – Programme Rendu Monnaie avec poids

Ici P est la liste des poids des pièces. On remarquera que dans notre matrice, ce sont des couples qui sont stockés : la valeur du rendu restant ainsi que le poids total de pièce utilisé jusqu'ici.

3.4 Comparaison avec un algorithme glouton

Pour résoudre ce problème, nous pouvons également utiliser un algorithme glouton qui va tout d'abord créer une liste avec des triplets en triant la liste correspondant aux poids, valeurs de la pièce et au rapport des deux. Puis, pour rendre la monnaie, le programme va parcourir la liste jusqu'à avoir une pièce inférieure au rendu et utiliser cette pièce.

Voici une implémentation de ce programme :

```
def Poids_Gloutonne(S,P,M):
    L=[(P[i]/S[i],S[i],P[i]) for i in range(len(S))]
    for i in range(len(L)):
        aux=L[i][0]
        indice=i
        for j in range(i+1,len(L)):
            if aux>L[j][0]:
                aux=L[j][0]
                indice=j
        L[indice],L[i]=L[i],L[indice]
    M1=M
    res=0
    while M1!=0:
        indice=0
        while indice < len(L) and M1<L[indice][1]:
            indice+=1
        r,s,p=L[indice]
        res =res + p*(M1//s)
        M1=M1%s
    return res
```

FIGURE 8 – Programme glouton Rendu Monnaie avec poids

Le programme glouton possède deux inconvénients. Le premier est qu'il est plus long que le programme en programmation dynamique. Le second est qu'il ne donne pas toujours la solution optimale. En effet, pour des systèmes de monnaie non canonique, le programme glouton ne donnera pas toujours la bonne valeur. Le second système est justement un système de monnaie non canonique. Nous avons implémenté une boucle pour savoir à partir de quel rendu les deux programmes avait une sortie différente :

```
S=[1,3,4,7]
P=[10,27,32,55]
compteur=1
a=Poids_Gloutonne(S,P,compteur)
b=Monnaie_Poids(S,P,compteur)[1]
while a==b and compteur<=20:
    compteur +=1
    a=Poids_Gloutonne(S, P, compteur)
    b=Monnaie_Poids(S,P,compteur)[1]
print(a,b,compteur)
```

FIGURE 9 – Comparaison des résultats des deux programmes

Et nous avons eu en sortie :

```
[4, [1, 1, 1, 4], [7, 0, 1]]
65 64 8
```

FIGURE 10 – Sortie de la comparaison

Ainsi, c'est pour un rendu de 8 avec ce système que l'algorithme glouton ne donne plus le bon résultat !

4 Conclusion

Pour conclure, nous avons vu comment utiliser la programmation dynamique pour résoudre le problème de rendu de pièce ainsi que des potentielles améliorations. Puis nous nous sommes intéressés au problème de rendu de monnaie avec minimisation de poids en utilisant la programmation dynamique et en la comparant à un algorithme glouton.