ENSSAT
LANNION

UNIVERSITÉ DE RENNES 1

Compte-rendu de mini-projet
Algorithmique Avancée

Étudiants :
Marc VACQUANT
Lilian SAVONA

Encadrants :
Cheikh Brahim El Vaigh

Sommaire

Introduction

Le projet s'est déroulé dans le cadre de la formation d'ingénieur informatique de deuxième année dispensée par l'ENSSAT.

Le sujet de ce projet est l'algorithmique avancée. Il s'agit d'un travail en collaboration par Lilian Savona et Marc Vacquant.

Le problème ici étudié est celui du rendu de monnaie. Le but est d'étudier la résolution de celui-ci à travers les algorithmes d'essais successifs, dynamiques, et gloutons.
Le sujet du projet peut être trouvé sous format pdf dans le dossier **Doc** sous la dénomination **Mini-projet_Algo-av_20-21.pdf**

Dans un premier temps, avant de passer à la réflexion des différents algorithmes nous avons définie une classe objet CoinList servant à manipuler le problème. Celle-ci est composée de deux tableaux de flot nommée valueList et coinList ainsi que d'un float totalValue, les deux tableaux correspondent respectivement à la liste des différentes pièces disponibles et le second à une liste de pièces, le float correspond à la valeur totale de cette liste de pièces.

Il est important de noter que les codes implémentés le sont avec la considération qu'une solution existe.

1. Essais successifs

La méthode des essais successifs consiste à avoir un vecteur candidat (taille, contenu) et d'essayer toutes les combinaisons possibles afin de trouver le meilleur candidat. Un candidat est formé d'une valeur *taille* qui correspond pour notre problème au nombre de pièces de la solution, et *contenu* la liste des pièces de la solution.

L'algorithme générique est notamment composé de *satisfaisant, enregistrer, soltrouvée,* et *défaire*.

satisfaisant : vérifie si l'ajout de pièce rapproche de la solution, les critères d'élagage sont également incorporés dans celle-ci. L'ajout de la pièce ne doit pas dépasser la valeur ciblée et on peut égaler en abandonnant la poursuite d'une solution si le nombre de pièces actuel est supérieur au nombre de pièces du meilleur candidat.

enregistrer : sauvegarde la solution actuelle en tant que meilleur candidat.

soltrouvée : informe sur le fait que la liste de pièces actuelle est solution ou non.

défaire : retire la dernière pièce de la liste des pièces afin de passer d'autres essais.

La condition d'élagage permet de limiter le nombre de calculs fait, couplé avec un tri décroissant de la liste des valeurs de pièces en début de recherche, on a comme première solution une solution qui correspond à celle de l'algorithme glouton. La solution de l'algorithme glouton n'est pas nécessairement optimale (comme nous le verrons dans la partie 3) cependant, il s'agit d'une solution relativement rapide à obtenir et qui de plus à l'avantage de fournir une solution assez proche d'une solution optimale. Cela permet surtout d'obtenir rapidement une première solution qui nous permettra d'arrêter un calcul dès que le nombre de pièces dépasse le nombre correspondant à la solution connue.

En ce qui concerne la complexité, on commence avec une liste vide et on fait card(C) appels, et pour chacun de ses appels, on refait card(C) appels tant que la liste satisfait les critères.Ainsi pour une liste de n, pièces il y aurait $\sum$ card(C)**k appel avec k appartenant à]0,n] soit une complexité maximale de O(card(C)**n).

Au maximum le nombre de pièces dans une solution est N (si l'on considère la pièce la plus petite ayant une valeur de $1)_1$ lorsque la solution est uniquement composée de pièces de valeur minimum. Si l'on explore toutes les possibilités à N/ éléments la complexité est O(card(C)**(N)

L'algorithme en pseudo code qui a été traduit en Java (package A) est le suivant(le document texte associé se trouve avec le code Java dans le dossier *Doc*.)

L'algorithme a été testé pour les valeurs 0; 45; et 54.54. en utilisant le système de pièces suivant: {2, 1, 0.50, 0.20, 0.10, 0.05, 0.02, 0.01}

Les solutions obtenues sont :

pour 0 : {}
pour 45 : {13,5,2,2,0.50,0.22,0.10,0.02,0.02}
pour 54.54 : {2.0, 2.0, 0.5, 0.02, 0.02}

Les solutions pour 45 et 54.54 ont été extrêmement longue à obtenir, ce n'est pas une méthode à utiliser.

```
//Variables

liste flottant : solution //initialisé à vide, meilleure solution actuelle
liste flottant : listeValeur // liste des valeurs de pièces
liste flottant : listePiece //initialisé à vide
flottant : objectif // valeur cherchée

//Fonctions

Fonction resoudre(liste flottant listePiece):


// on boucle sur les valeurs de pièces
Pour i dans listeValeur:

    // on ajoute la pièce actuelle à la liste
    ajoutPiece(i,listePiece)

    // on vérifie le caractère satisfaisant de la nouvelle liste, satisfaisant contient les conditions d'élagage
    Si satisfaisant(listePiece):

        // on vérifie si la nouvelle liste est solution
        Si soltrouvée(listePiece):

            // si la nouvelle liste est solution on vérifie si elle est meilleure solution si il y avait une solution connu
            Si taille(solution)>taille(listePiece) ou si solution est vide:

                // on enregistre la solution si elle est meilleure ou si c'est la première solution trouvé
                enregistrer(listePiece)

            FinSi

        FinSi

        // si la liste n'est pas solution
        Sinon:

            // on appel resoudre avec la nouvelle liste
            resoudre(listePiece)
            // on retire la pièce ajouté pour procéder à la suite des combinaisons
            défaire(listePiece)
        FinSinon
    FinSi
    Sinon
        // on retire la pièce ajouté pour procéder à la suite des combinaisons
        défaire(listePiece)
    FinSinon
FinPour
```

Pseudo-code de l'algorithme d'essais successifs

2. Programmation dynamique

On appelle NBP(*i*, *j*) le nombre minimal de pièces nécessaires pour payer la somme *j* en ne s'autorisant que le sous-ensemble des pièces $\{c_1, \ldots, c_i\}$

Sa formule de récurrence peut être énoncée ainsi :
$NBP(i, 0) = 0$
$NBP(i, j) = 1 + min_{1 \leq k \leq i} (NBP(i, j - c_k))$

La première ligne signifie que le nombre minimal de pièces nécessaires pour payer la somme 0 est 0, peu importe le sous-ensemble de pièces utilisé.

La seconde ligne de notre récurrence permet d'appliquer le principe de la programmation dynamique. Pour composer la solution optimale de notre problème, nous allons utiliser les valeurs optimales associées à des sous-problèmes.
Ainsi, pour calculer le nombre minimal de pièces nécessaires pour payer la somme j, il faut trouver la solution optimale quand on veut trouver le minimum de pièces nécessaires pour *j* soustrait par les différentes pièces disponibles.
Pour cela, il faut prendre 1 et ajouter la plus petite valeur parmi le calcul NBP pour la somme soustraite des différentes pièces disponibles dans le sous-ensemble $\{c_1, \ldots, c_i\}$

Pour résoudre ce problème, nous allons prendre une approche où nous calculerons "de bas en haut".

Nous pouvons créer un tableau de $j + 1$cases allant de 0 à j.
Les cases sont initialisées à $j + 1$ou à une valeur strictement supérieure à j. La première case d'indice 0 est initialisée à 0.
Chaque case contient le nombre de pièces de monnaie nécessaire pour rendre la somme représentée par l'indice de la case.

0	1	...	j - 1	j
0	j + 1	j + 1	j + 1	j + 1

L'évolution du calcul permettra de remplir les différentes cases du tableau en observant pour chaque pièce de l'ensemble $C = c_1, \ldots, c_n$ s'il est possible de rendre moins de pièces que la valeur présente dans la case.

Pour donner un exemple de cette évolution, nous pouvons le présenter avec la somme 6 et le sous-ensemble de pièces C = {1, 3, 4}

0	1	2	3	4	5	6
0	j + 1	j + 1	j + 1	j + 1	j + 1	j + 1

Pour rendre la somme 1, nous pouvons uniquement utiliser la pièce 1 de C.
On indique ainsi que le nombre de pièces minimum pour rendre la valeur 1 est 1.

0	1	2	3	4	5	6
0	1	j + 1	j + 1	j + 1	j + 1	j + 1

Pour rendre la somme 2, nous pouvons uniquement utiliser la pièce 1 de C.
Trouver la solution de 2 revient à trouver la solution de 1 en ajoutant une pièce.
On indique ainsi que le nombre de pièces minimum pour rendre la valeur 2 et est 2

0	1	2	3	4	5	6
0	1	2	j + 1	j + 1	j + 1	j + 1

Pour rendre la somme 3, nous pouvons utiliser les pièces 1 et 3 de C.
Si on utilise la pièce 1, on peut trouver la solution en s'aidant du sous-problème de rendre 2 que nous avons développé à l'étape précédente. Nous obtenons ainsi que l'on peut rendre 3 en 3 pièces.
Si on utilise la pièce 3, on arrive directement à 0. Ce qui nous donne un résultat en une pièce, ce qui est plus désirable.
On indique ainsi que le nombre de pièces minimum pour rendre la valeur 3 est 1

0	1	2	3	4	5	6
0	1	2	1	j + 1	j + 1	j + 1

En continuant ces calculs, nous pouvons arriver au résultat suivant :

0	1	2	3	4	5	6
0	1	2	1	1	2	2

La résolution de notre problème nous indique que la valeur 6 peut être rendue en 2 pièces.

Afin de savoir quelles pièces rendre en plus, nous pouvons ajouter une ligne à notre tableau afin de retenir la pièce utilisée lors du passage d'un sous-problème à un autre. Nous obtiendrons le résultat suivant pour l'exemple précédent.

0	1	2	3	4	5	6
0	1	2	1	1	2	2
0	1	1	3	4	1	3

Pour passer du sous-problème de rendu de monnaie de 2, nous utilisons une pièce de 1, ce qui nous amène au sous-problème de rendu de monnaie de 1.
Pour passer du sous-problème de rendu de monnaie de 1, nous utilisons une pièce de 1, ce qui nous amène au sous-problème de rendu de monnaie de 0, la fin de notre récursion.
Nous pouvons ainsi utiliser cette nouvelle ligne pour connaître les pièces nécessaires pour rendre la monnaie. Dans le cas de la valeur 2, il faut deux pièces de 1.

Nous avons les complexités suivantes avec *j*, la somme à rendre et *i,* le total de pièce disponibles dans le sous-ensemble$\{c_1, \ldots, c_i\}$

Complexité en temps : O(*j* * *i*)
Pour chaque montant *j* du tableau, nous allons potentiellement calculer *i* valeurs avec les pièces de notre sous-ensemble.

Complexité en espace: O(j)
Nous répondons et stockons un total de *j* sous-problèmes dans notre tableau de programmation dynamique pour arriver à notre réponse optimale.

En reprenant les valeurs et le sous-ensemble de pièces de la partie sur les essais successifs, nous obtenons les résultats suivants :

Pour la somme 0, la solution est [0, 0, 0, 0, 0, 0, 0, 0]

Pour la somme 45, la solution est [22, 1, 0, 0, 0, 0, 0, 0]
Ce qui correspond à 22 pièces de deux euros et une pièce d'un euro.

Pour la somme 54.54, la solution est [27, 0, 1, 0, 0, 0, 2, 0]
Ce qui correspond à 27 pièces de deux euros, une pièce de 50 centimes et deux pièces de 2 centimes.

```
// Variables
liste flottant : coinList              // Liste des valeurs de pièces
liste entier : tableMem                // Mémorise le nombre de pièces nécessaires pour chaque sous-problème
liste entier : coinUsed                // Mémorise la pièce utilisée pour le passage à un autre sous-problème
liste entier : coinListEntier          // Pièce sous la forme de centimes pour ne pas avoir de flottant
entier : sommeEnCentime                // Somme en centime afin de pouvoir l'utiliser en indice de tableau
entier : reponsePotentiel              // Nouvelle solution potentielle d'un sous-problème
entier : index                         // Permet de retrouver les pièces utilisées pour résoudre le problème
entier : coinIteration                 // Variable servant lors de l'itération pour retrouver les pièces utilisées


// Fonction
Fonction resoudre(listePiece, somme):
        sommeEnCentime = somme*100
        // On remplit le coinListEntier afin d'avoir des centimes (entier) au lieu d'avoir des flottants
        Pour i allant de 0 à (taille(coinList) - 1)
                coinListEntier[i] <- (coinList[i] * 100)
        Fin Pour

        // Le montant minimum de pièces à utiliser pour faire de la monnaie pour 0 est 0
        tableMem[0] <- 0
        coinUsed[0] <- 0

        // On initialize toutes les cases du tableau tableMem à une valeur plus grande que l'objectif
        Pour i allant de 0 à sommeEnCentime
                tableMem[i] <- (sommeEnCentime + 1)
        Fin Pour

        // Calcule le nombre de pièces minimum requis pour toutes les valeurs allant de 1 à sommeEnCentime
        // Pour chaque case de tableMem, on mémorise également les pièces à rendre
        Pour i allant de 0 à sommeEnCentime:
                Pour k dans coinListEntier
                        reponsePotentiel <- 1 + tableMem[i - l]
                        tableMem[i] <- Minimum(tableMem[i], reponsePotentiel)
                        // Si la réponse potentiel était inférieur à tableMem[i], on enregistre la pièce utilisée
                        Si (tableMem[i] >= reponsePotentiel) Alors
                                coinUsed[i] <- k
                        Fin Si
                Fin Pour
        Fin Pour

        // Affichage du nombre de pièces minimum nécessaires
        afficher("Nombre minimum de pièces : ", tableMem[sommeEnCentime])
        // Affichage des pièces utilisées
        index = sommeEnCentime
        coinIteration = coinUsed[sommeEnCentime]
        afficher("Pièces utilisées")
        Tant que (index != 0)
                afficher(coinIteration/100)
                index <- index - coinIteration
                coinIteration <- coinUsed[index]
        Fin Tant que
Fin
```

Pseudo-code de l'algorithme de programmation dynamique

3. Algorithme glouton

Nous avons codé un algorithme glouton pour le problème, celui-ci se trouve dans le package C, il est important de noter que celui-ci peut ne pas aboutir sur une solution s'il n'existe pas une pièce d'unité (de valeur 1 ou 0.01 pour les monnaies en valeurs entières ou décimales.

Avec le système suivant:

$C = \{c_1, c_2, c_3\}$, $d_1 = 6$, $d_2 = 4$, $d_3 = 1$ et $N = 8$.

L'algorithme glouton renvoie la solution {6,1,1}, car il résout le problème en mettant les pièces de plus haute valeur en priorité.

Or ici la solution optimale est {4,4}

Pour l'euro, il n'y a pas de valeur entre deux pièces pour laquelle un résultat autre que la solution gloutonne est meilleure (soit toutes les valeurs entre 0 et 2€ ont une solution optimale avec l'algorithme glouton) puis il faut vérifier les valeurs entre la pièce la plus haute et son double, car ainsi toute valeur supérieure pourra s'exprimer comme une combinaison linéaire de pièces inférieures à la plus grande modulo la plus grande pièce:

$N = CL \bmod(c_1)$

Où N est la valeur cherchée, CL une combinaison linéaire optimale pour une valeur inférieure à c_1.

De cette manière on peut montrer que C'= {50, 30, 10, 5, 3, 1} est un système pour lequel l'algorithme glouton est optimal.

Suite à des recherches, un algorithme glouton est optimal pour un système de pièces qualifié de système canonique. Selon une démonstration de D. Kozen et S. Zaks vérifier que les valeurs v tel que $c_1+1<v<c_1+c_2$ suffit à démontrer qu'un système est canonique.

L'algorithme a été testé pour les valeurs 0; 45; et 54.54. en utilisant le système de pièces suivant: {2, 1, 0.50, 0.20, 0.10, 0.05, 0.02, 0.01}

Les solutions obtenus sont :

pour 0 : {}

pour 45 : {13,5,2,2,0.50,0.22,0.10,0.02,0.02}

pour 54.54 : {2.0, 2.0, 0.5, 0.02, 0.02}

Les solutions sont bien plus rapides que pour l'algorithme par essais successifs, mais cela fonctionne bien car le système de pièces utilisé est canonique.

4. Questions complémentaires

1) Les algorithmes ont été implémentés avec une conception orientée objet, ainsi l'algorithmes A en particulier est très peu efficace car il y a création d'un objet CoinList à chaque appel de la fonction resolveRec alors que simplement créer une nouvelle liste et conserver les paramètres tels que la liste des valeurs dans l'objet AlgoGen qui est la résolution limiterait la prise d'espace inutile.

2) Il serait possible de résoudre ce problème avec une solution de type "Diviser pour régner" mais l'algorithme risquerait de calculer plusieurs fois le même sous-problème au lieu de l'enregistrer comme le fait l'algorithme de programmation dynamique. La solution serait ainsi moins efficace et aurait une complexité en temps moins intéressante.

3) Le problème peut être résolu en mélangeant les algorithmes à essais successifs et gloutons. En utilisant en premier lieu un algorithme glouton, on obtient dans un premier temps un résultat peu éloigné de la solution optimale (cela dépend du système testé et de la valeur) puis on utilise l'algorithme à essais successifs en utilisant la solution gloutonne comme solution au début limitant donc les nombres d'appels avec l'élagage sur le nombre de pièces. De plus tester les pièces dans un ordre décroissant limite des solutions nécessairement plus mauvaise :
Tester les combinaisons avec les pièces de plus hautes valeurs a plus de chance de résulter en l'obtention d'une meilleure solution, réduisant la taille de la solution et élaguant des solutions nécessairement plus grandes. On peut encore améliorer l'algorithme en ajoutant une condition d'élagage supplémentaire : pour un état donné où il reste au plus k pièces pour avoir un meilleur résultat si la valeur actuelle plus k fois la valeur de la plus haute pièce est inférieur à l'objectif alors on ne peut pas obtenir un meilleur candidat avec ce début de solution, on peut donc couper court à la recherche avec ce fragment de solution.

L'algorithme dynamique garde une complexité en temps et en espace plus intéressante que l'algorithme d'essais successifs seul. Il est cependant moins intéressant que l'algorithme glouton dans le cas où le sous-ensemble de pièces est canonique.

Pour résoudre le problème RLMMO, il vaut mieux choisir l'algorithme dynamique si le système de monnaie n'est pas canonique et l'algorithme glouton si le système est canonique.

L'algorithme glouton aura une complexité en temps et en espace moins importante mais ne pourra pas toujours trouver la meilleure solution avec un système non-canonique.

Conclusion

Les trois types d'algorithmes ne sont pas équivalents. Comme attendu, les essais successifs bien que donnant nécessairement la meilleure solution n'est pas une méthode envisageable pour une question de temps de calculs. L'algorithme glouton est une solution intuitive et rapide même si elle n'est pas nécessairement optimale. En revanche l'algorithme dynamique fournit une réponse optimale et est plus rapide que les essais successifs mais moins que l'algorithme glouton.

Ainsi pour une implémentation il faut choisir en fonction de notre connaissance du système:

L'algorithme dynamique requiert beaucoup de place et est utilisé pour les systèmes non-canoniques ou inconnus.

L'algorithme glouton est pour les systèmes canoniques ou lorsque une réponse non optimale est viable et qu'il y a un problème de place.

L'algorithme d'essais successifs n'est à utiliser que si vous avez besoin de transformer votre ordinateur en radiateur pour votre pièce.

Cependant pour la vie de tous les jours où nous devons calculer de nous même avec le système de l'euro, l'algorithme glouton reste le mieux car il est optimal et surtout bien moins demandant.