ENSSAT

Informatique 2ème année

Compte-rendu Mini Projet Algorithme Avancée

(Langage : Java)

Mathias Penavaire

Mathias Peuch

14/04/2024

**Introduction :**

Ce projet s’intéresse au problème du rendue de monnaie. Un sujet présent dans la vie de tous les jours : quand nous achetons notre baguette de pain ou faisons nos courses en payant en espèces. Il existe une multitude possibilité pour rendre une somme X en euros, mais est-il vraiment efficace et souhaitable de rendre par exemple, 1€ en pièces uniquement de 1 centimes ?

Nous allons rappeler ici quelques notations (celles du sujet) important pour la bonne compréhension de la suite du compte-rendu :

* C : l’ensemble des pièces de monnaie disponibles. En France, il y a 8 pièces (2€, 1€, 50cts, 20cts, 10cts, 5cts, 2cts, 1cts). C = {200,100,50,20,10,5,2,1}. Remarque : dans les programmes nous avons choisi de faire les calculs en centimes
* RLMMO : rendre la monnaie de manière optimale, c’est-à-dire, rendre en un minimum de pièces possibles. Si l’on veut rendre 1€20, on prend une pièce de 1€ et une pièce de 20cts.
* N : ce qu’il faut rendre. Par exemple, si le client nous as donné 10€ et qu’il devait payer 9€, N vaut ici 1€.
* S : l’ensemble solutions qui contient les pièces. Il est nécessaire que la somme de ses éléments soient égale à N (le but est de rendre ce qu’il faut, ni plus, ni moins).

Pour répondre au problème RLMMO, nous allons nous intéresser à 3 façons de procéder vues en cours. Nous commencerons par la méthode des essais successifs, puis nous passerons à la programmation dynamique et enfin à l’algorithme glouton. Nous finirons par une conclusion présentant notre avis sur les différentes méthodes.

1. **Essais successifs :**

Dans cette partie, nous allons utiliser la méthode des essais successifs, également connue sous le nom de méthode de force brute. En bref, elle consiste à résoudre un problème en essayant toutes les possibilités, souvent de manière systématique.

Avant de regarder de plus près aux constituants de l’algorithme générique (satisfaisant, enregistrer, soltrouvée, défaire), intéressons nous au vecteur représentant un candidat.

* Taille : T>0 car N>0. En effet, si on note T la taille du vecteur candidat S, alors elle est strictement positive car ici N est strictement positive. Il faut forcément rendre au moins 1 pièce au client.
* Contenu : S = Tableau de T valeurs dont les éléments sont dans C (T représentant la taille). Les éléments correspondent aux pièces à rendre et le nombre est caractérisée par T.

Maintenant que nous avons caractérisé le vecteur candidat, passons à l’algorithme générique.

On utilisera une variable Reste initialisée à N.

Satisfaisant : à étape i si (on a S = {ck,…,ck+(i-2)} qui est satisfaisant à étape i-1) :

* Reste => 0
* Ci<=Reste

Il faut qu’il y ait une valeur non nulle à rendre et que la pièce choisit soit inférieure au reste restant, sinon on rendra trop ce qui n’est pas voulu.

Enregistrer : à l’étape i, S[i] <- cj

Reste = Reste – dj

On met l’élément cj à l’indice i dans le vecteur candidat.

On met à jour le reste en enlevant la valeur de la pièce cj.

Soltrouvé : à étape i, Reste = 0

Si le reste est nul, on a trouvé un ensemble S dont la somme de ses éléments vaut N.

Défaire(i) : L’élément d’indice i est libéré de S

Reste = Reste + dj (dj représente la valeur de la pièce cj à l’indice i)

On enlève l’élément en question du vecteur candidat S et on ajoute au reste sa valeur.

**Procédure algorithme générique :**

procédure pieces (ent i) ;

var ent Reste = N\*100 ; # Nous travaillons en cts d’euros

var tab piecesDisponibles = permutation([200,100,50,20,10,5,2,1]) # La permutation permet d’avoir une solution quelconque et pas la solution optimale 200 = 200cts€ = 2€ …

début

# Parcourt de la liste des pièces

pour ci dans piecesDisponibles faire

# Satisfaisant

si Reste=>0 et ci<=Reste alors

# Enregistrer

S[i]<-ci ;

Reste -= di ;

# Soltrouvée ?

si Reste = 0 alors écriresolution

# TouteSol(i+1)

sinon pieces(i+1)

fsi

# Défaire choix

Reste += di ;

enlever(S[i]);

Fsi

Fait

Fin

Nous avons choisit de faire 3 programmes en Java : ToutesSol.java, SolOpt.java et SolRapide.java.

Le premier va faire comme la procédure de l’algorithme générique, il va parcourir toutes les solutions « bêtement », le deuxième va ne garder que les éléments meilleurs que l’ancienne meilleure solution (cf : « sol\_pot.size() <= solution.size() ». sol\_pot est le vecteur candidat S et solution est l’actuel meilleur vecteur candidat. Le meilleur est celui qui a la taille la plus petite. Le troisième fait comme le deuxième, sauf qu’il utilise une condition d’élagage, qui réduit considérablement le nombre de recherches (voir parties sur l’efficacité).

La condition d’élagage est la suivante : « else if (sol\_pot.size() <= solution.size())[…] appel de SolRapide(i+1) ». En faisant cela, nous évitons de faire des appels inutiles. En effet, c’est inutile de chercher un vecteur candidat s’il dépasse déjà la taille du meilleur vecteur solution.

**Procédure algorithme solution optimale :**

procédure pieces (ent i) ;

var ent Reste = N\*100 ; # Nous travaillons en cts d’euros

var ent compteur = 0 ; # Compteur permettant de compter le nombre d’appels récursifs

var tab solution = [1…1] ; # Liste contenant la solution finale, initialisée avec n « 1 », n représentant le reste

var tab sol\_pot = [] ; # Vecteur candidat

var tab piecesDisponibles = [200,100,50,20,10,5,2,1] ; # Liste de pièces

début

# Parcourt de la liste des pièces

pour ci dans piecesDisponibles faire

# Satisfaisant

si Reste=>0 et ci<=Reste alors

# Enregistrer

Sol\_pot[i]<-ci ;

Reste -= di ;

# Soltrouvée  et meilleur

si Reste = 0 et taille(sol\_pot)<taille(solution) alors

solution <- sol\_pot ;

# TouteSol(i+1)

sinon

compteur += 1 ;

pieces(i+1)

fsi

# Défaire choix

Reste += di ;

enlever(sol\_pot[i]);

Fsi

Fait

Fin

**Procédure algorithme solution rapide :**

procédure pieces (ent i) ;

var ent Reste = N\*100 ; # Nous travaillons en cts d’euros

var tab solution = [1…1] ; # Liste contenant la solution finale, initialisée avec n « 1 », n représentant le reste

var ent compteur = 0 ; # Compteur permettant de compter le nombre d’appels récursifs

var tab sol\_pot = [] ; # Vecteur candidat

var tab piecesDisponibles = [200,100,50,20,10,5,2,1] ; # Liste de pièces

début

# Parcourt de la liste des pièces

pour ci dans piecesDisponibles faire

# Satisfaisant

si Reste=>0 et ci<=Reste alors

# Enregistrer

Sol\_pot[i]<-ci ;

Reste -= di ;

# Soltrouvée  et meilleur

si Reste = 0 et taille(sol\_pot)<taille(solution) alors

solution <- sol\_pot ;

# TouteSol(i+1)

sinon

# Encore Possible

Si taille(sol\_pot)<taille(solution) alors

Compteur += 1 ;

pieces(i+1) ;

fsi

# Défaire choix

Reste += di ;

enlever(sol\_pot[i]);

Fsi

Fait

Fin

La variable compteur utilisée dans les procédures SolRapide et SolOptimale permet de montrer l’efficacité de la condition d’élagage choisit. Dans les programmes en annexe, on a choisit de faire un print du compteur et on peut voir une différence significative même pour des petites valeurs de N.

Nous avons essayé avec N=10 et SolRapide faisait 3 appels, tandis que SolOptimale en faisait plus de 50.

Si nous essayons avec de plus grandes valeurs de N, ce qui est souvent le cas dans la vraie vie, il est donc préférable d’avoir un système qui utilise SolRapide.s

1. **Programmation dynamique :**

Soit NBP(i, j) le nombre minimal de pièces nécessaires pour payer la somme j, en ne s’autorisant que le sous-ensemble des pièces {c1, . . . , ci}. Voici la formule de récurrence :

NBP(i,0) = 0 (1)

NBP(i,j) = 1 + NBP(i,j-c\_i) si j=> c\_i (2)

NBP(i,j) = NBP(i-1,j) sinon (3)

1. : Le nombre minimal de pièces pour rendre 0€ est 0 (cas de base)
2. : Si l’on doit rendre une valeur (reste) supérieure à la plus grande pièce alors on ajoute 1 et on met à jour la valeur de j.
3. : Si j est inférieur à la plus grande pièce, alors on enlève 1 à i. On va prendre ensuite une pièce plus petite.