Programmation dynamique

Technique algorithmiques

- « brute-force »
- récursion
- retour en arrière (*backtracking*)
- diviser pour régner
- algorithmes gloutons (Dijkstra, …)
- **programmation dynamique**
- séparation et évaluation (*branch and bound*)
- algorithmes randomisés

Problème du rendu de monnaie

- *Problème* : étant donné un système de monnaie (dénomination de pièces), décomposer le montant de *x* de façon optimale (nombre minimum de pièces)

- Pour les systèmes de monnaie *canoniques* l'algorithme glouton produit la solution optimale
  - 29c=20+5+2+2
  - si l'on avait la pièce de 14c, ca ne marcherait pas : 29c=14+14+1

- trouver la décomposition optimale pour un système de monnaie *arbitraire*

- pour illustrer la suite : rendre 63 avec les pièces (1,5,10,21,25)

Première solution

- On suppose toujours qu'il existe la pièce de 1c
- Pour décomposer *K* centimes :
  - s'il existe la pièce de *K*, utiliser-la
  - sinon pour tout *i*<*K*
    - calculer une décomposition optimale pour *i*
    - calculer une décomposition optimale pour *K-i*
  - choisir *i* qui minimise la somme
- Cet algorithme peut être vu comme diviser-pour-régner ou brute-force
  - la solution est récursive
  - l'algorithme est exponentiel
  - il prend beaucoup de temps et il est souvent infaisable

Autre solution

• Réduire le calcul en choisissant la première pièce et résolvant le reste
• Pour 63c :
  • pièce de 1c plus une solution pour 62c
  • pièce de 5c plus une solution pour 58c
  • pièce de 10c plus une solution pour 53c
  • pièce de 21c plus une solution pour 42c
  • pièce de 25c plus une solution pour 38c
• Choisir la meilleur solution parmi les cinq
• Au lieu de résoudre récursivement 62 problèmes on en résout 5
• Mais ça reste coûteux (exponentiel en terme de nombre de sous-problèmes à résoudre)

Programmation dynamique

• *Idée* : Résoudre le problème pour 1c, 2c, 3c, … jusqu'au montant désiré. Sauvegarder le résultat (nb de pièces) dans un tableau. Pour un nouveau montant *N* utiliser les réponses précédentes.

• Par ex. pour calculer une solution pour 13c :
  • résoudre le problème pour 1c,2c,3c,…,12c
  • choisir la meilleure solution parmi les suivantes :
    • pièce de 1c + solution optimale pour 12c
    • pièce de 5c + solution optimale pour 8c
    • pièce de 10c + solution optimale pour 3c

Programmation dynamique

• Le premier et le deuxième algorithmes sont récursifs ➠ arbre d'appels récursifs ➠ nombre exponentiels de sous-problèmes à résoudre

• L'algorithme de programmation dynamique prend un temps $O(N \cdot K)$ où N est le montant et K le nombre de dénominations de pièces

Comparaison avec *diviser-pour-régner*

• *Diviser-pour-régner* divise le problème en sous-problèmes indépendants, résout-les et combine les solution en une solution du probème de départ
• *Diviser-pour-régner* est une approche *top-down*
• Programmation dynamique calcule des solutions de petits sous-problèmes pour les combiner en une solution d'un problème plus grand. C'est donc une approche *bottom-up*.
• Contrairement à *Diviser-pour-régner,* les sous-problèmes sont *dépendants*
• *Principe d'optimalité* : solution optimale d'un problème contient des solutions optimales de toutes les sous-problèmes
• Il reste de trouver une décomposition en sous-problèmes
• Les solutions des sous-problèmes sont mémorisées dans un tableau

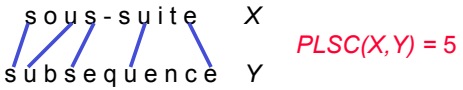
La plus longue sous-séquence commune à deux séquences

Sous-séquence

• mot (séquence) *Y* est une sous-séquence de *X* si *Y* peut être obtenu de *X* en effaçant certaines lettres

• sous-séquence (*subsequence*) = sous-mot, sous-suite

• *Problème* : étant donné deux mots, trouver une sous-séquence commune dont la longueur est maximale



Distance

• $d(X,Y)=|X|+|Y|-2\cdot PLCS(X,Y)$
distance entre *X* et *Y* : nombre d'insertions/suppressions de lettres nécessaire pour transformer *X* en *Y*

• d'autres distances :
  • distance d'édition (Levenshtein) : substitutions/insertions/suppressions
  • distance de Hamming : substitutions

• Applications :
  • comparaison (alignement) de séquences biologiques (AND,protéines)
  • comparaison de fichiers texte (commande `diff` sous Linux)

Solution naive

• pour chaque sous-séquence de *X* vérifier si elle est sous-séquence de *Y*
• il y a 2^n sous-séquences de *X* (*n* longueur de *X*) ➠ algorithme exponentiel

Principe d'optimalité

UMLV ©

Notation : étant donné $X=x_1x_2\ldots x_m$, $X_k=x_1x_2\ldots x_k$ (préfixe de longueur k)

Théorème : Soit $X=x_1x_2\ldots x_m$ (= X_m) et $Y=y_1y_2\ldots y_n$ (= Y_n). Soit $Z=z_1z_2\ldots z_k$ (= Z_k) une sous-séquence la plus longue de X et Y.

- si $x_m = y_n$, alors $z_k = x_m = y_n$, et Z_{k-1} est une PLSC de X_{m-1} et Y_{n-1}
- si $x_m \neq y_n$,alors $z_k \neq x_m$ implique que Z est une PLSC de X_{m-1} et Y_n
- si $x_m \neq y_n$, alors $z_k \neq y_n$ implique que Z est une PLSC de X_m et Y_{n-1}.

Solution récursive

UMLV ©

Le théorème implique une décomposition du problème en sous-problèmes :

- si $x_m = y_n$, calculer PLSC de X_{m-1} et Y_{n-1}, puis ajouter x_m.
- si $x_m \neq y_n$, calculer PLSC de X_{m-1} et Y_n ainsi que PLSC de X_m et Y_{n-1}, puis choisir la plus longue des deux

A noter la structure "chevauchante" des sous-problèmes :
Les PLSC de X_{m-1} et Y_n et de X_m et Y_{n-1} démandent une solution de PLSC de X_{m-1} et Y_{n-1}

Soit $c[i,j]$ la longueur de la PLSC de X_i et Y_j .

$$c[i,j]=\begin{cases} 0 & \text{si } i=0, \text{ ou } j=0 \\ c[i-1,j-1]+1 & \text{si } i,j>0 \text{ et } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{si } i,j>0 \text{ et } x_i \neq y_j \end{cases}$$

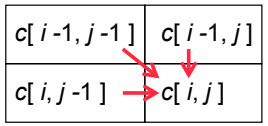
Algorithme de calcul de *c*

UMLV ©

$$c[i,j]=\begin{cases} 0 & \text{si } i=0, \text{ ou } j=0 \\ c[i-1,j-1]+1 & \text{si } i,j>0 \text{ et } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{si } i,j>0 \text{ et } x_i \neq y_j \end{cases}$$

Calculer le tableau $c[0..m,0..n]$, où $c[i,j]$ est défini ci-dessus. $c[m,n]$ est alors la réponse (longueur de la PLSC).

Pour chaque entrée $c[i,j]$, garder la trace du calcul (une des trois flêches rouges) :



Exemple de matrice *c*

UMLV ©

PLSC("providence", "president")

i		j	0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0			0	0	0	0	0	0	0	0	0	0	0
1	p		0	1	1	1	1	1	1	1	1	1	1
2	r		0	1	2	2	2	2	2	2	2	2	2
3	e		0	1	2	2	2	2	2	3	3	3	3
4	s		0	1	2	2	2	2	2	3	3	3	3
5	i		0	1	2	2	2	3	3	3	3	3	3
6	d		0	1	2	2	2	3	4	4	4	4	4
7	e		0	1	2	2	2	3	4	5	5	5	5
8	n		0	1	2	2	2	3	4	5	6	6	6
9	t		0	1	2	2	2	3	4	5	6	6	6

Exemple de matrice *c*

UMLV ©

PLSC(“providence“, “president“)

i		j	0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0			0	0	0	0	0	0	0	0	0	0	0
1	p		0	1	1	1	1	1	1	1	1	1	1
2	r		0	1	2	2	2	2	2	2	2	2	2
3	e		0	1	2	2	2	2	2	3	3	3	3
4	s		0	1	2	2	2	2	2	3	3	3	3
5	i		0	1	2	2	2	3	3	3	3	3	3
6	d		0	1	2	2	2	3	4	4	4	4	4
7	e		0	1	2	2	2	3	4	5	5	5	5
8	n		0	1	2	2	2	3	4	5	6	6	6
9	t		0	1	2	2	2	3	4	5	6	6	6

917

Exemple de matrice *c*

UMLV ©

PLSC(“providence“, “president“)

i		j	0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0			0	0	0	0	0	0	0	0	0	0	0
1	p		0	1	1	1	1	1	1	1	1	1	1
2	r		0	1	2	2	2	2	2	2	2	2	2
3	e		0	1	2	2	2	2	2	3	3	3	3
4	s		0	1	2	2	2	2	2	3	3	3	3
5	i		0	1	2	2	2	3	3	3	3	3	3
6	d		0	1	2	2	2	3	4	4	4	4	4
7	e		0	1	2	2	2	3	4	5	5	5	5
8	n		0	1	2	2	2	3	4	5	6	6	6
9	t		0	1	2	2	2	3	4	5	6	6	6

918

Conclusions

UMLV ©

Algorithme prend un temps O(*nm*) où *n* et *m* sont les longueurs des séquences

Il demande un espace O(*nm*) également. Il existe un algorithme qui utilise un espace O(*n*) (Hirschberg 1975)

Le temps peut être amélioré à O(*nm*/log *n*) (Masek, Paterson 1983) (Crochemore&Landau&Ziv-Ukelson 2003)

919