Leçon 14 : Programmation dynamique

Niveau : Tle / MPII

Prérequis : Glouton ; Diviser pour régner

I - Principe

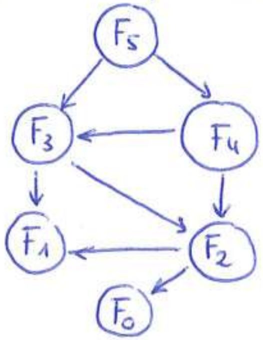
I.1 - Motivation

Algorithme 1 : Implémentation naïve de la suite de Fibonacci

```
def Fibo(n):
    if n == 0 or n == 1 :
        return n
    else :
        return Fibo(n-1) + Fibo(n-2)
```

Complexité : $O(2^n)$

Graphe de dépendance des sous-problèmes pour $n=5$:



Les sous-problèmes se chevauchent : la méthode diviser pour régner est inefficace.

En programmation dynamique, on stocke les valeurs des sous-pb pour éviter les recalculs.

Algorithme 2 : Fibonacci avec stockage

```
def Fibo(n):
    F = [0,1]
    for i in range(2, n+1):
        F.append(F[i-1] + F[i-2])
    return F[n]
```

I.2 - Définition

Définition 3 : La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes en stockant les résultats intermédiaires (principe de mémoïsation)

Mise en œuvre 4 :

① - Complexifier le pb en créant des sous-problèmes

② - Trouver une relation entre les sous-problèmes

③ - Résoudre les sous-problèmes en utilisant la relation :
- soit impérativement, du plus petit au plus grand, en remplissant un tableau des sous-pb (méthode ascendante)
- soit récursivement, en utilisant la mémoïsation (méthode descendante)

Remarque 5 : l'algorithme 2 utilise la méthode ascendante. On aurait pu utiliser uniquement deux variables et écraser les résultats intermédiaires.

Remarque 6 : Dans le paradigme "Diviser pour régner", les sous pb sont indépendants. On les résout donc récursivement sans mémoïser.

Remarque 7 : Pour obtenir, en plus de la valeur de la solution optimale, quelle est cette solution, on peut mémoriser quel(s) sous-pb on a utilisé pour l'obtenir.

Développement 1 : Illustration du paradigme sur le pb du chemin dans la pyramide.

Nom : Daphné Kany
Sujet : 14 Jury :

II - Algorithmes illustrant le principe

II.1 - Rendu de monnaie

Instance : n pièces $p_1, \dots, p_n \in \mathbb{N}^n$, $S \in \mathbb{N}$ la somme à rendre
Problème : trouver un n-uplet $T = (x_1, \dots, x_n) \in \mathbb{N}^n$ tq $\sum_{i=1}^{n} x_i p_i = S$ et qui minimise $\sum_{i=1}^{n} x_i$.
(ie trouver le nombre minimum de pièces pour rendre la monnaie).

Algorithme 8 : approche gloutonne
1- Ajouter la pièce p_i de plus grd valeur $\leq S$
2- Recommencer avec $S - p_i$

Cet algorithme est-il optimal ?

Exercice 9 : Avec les pièces (4,3,1), trouver une somme S pour laquelle le glouton n'est pas optimal
(S = 6 convient).

Programmation dynamique 10 :
① - On considère les sous-problèmes $R(s)$ pour $s \in [\![0, S]\!]$, le nombre minimum de pièces pour rendre s.
② - Pour trouver la relation de récurrence, on regarde la dernière pièce rendue p_i. Alors $R(S) = 1 + R(S - p_i)$. La dernière pièce $p_i \in \{p_1, \dots, p_n\}$, donc

$$R(S) = \begin{cases} \min_{p_i \in \{p_1, \dots, p_n\}} (R(S - p_i) + 1) & \text{si } S > 0 \\ 0 & \text{si } S = 0 \\ +\infty & \text{si } S < 0 \text{ (rendu impossible)} \end{cases}$$

Algo 11 : (Rendu de monnaie, methode descendante)

```
def rendu (P, S, m) : # P est un tableau tq P[i] = pi
    if S == 0 :        # m est un tableau de mémoisation
        return 0       # de taille S initialisé à 0
    if m[S] > 0 :
        return m[S]
    n = infinity
    for p in P :
        n = min (n, 1 + rendu (P, S-p, m))
    m[S] = n
    return n
```

Complexité : $O(nS)$

II.2 - Sac à dos

Instance : n objets de poids $\{w_1, \dots, w_n\} \in \mathbb{N}^n$, et de valeur $\{v_1, \dots, v_n\} \in \mathbb{N}^n$. Une capacité $W \in \mathbb{N}$.
Pb : Trouver $T = (x_1, \dots, x_n) \in \{0,1\}^n$ tq $\sum_{i=1}^{n} x_i w_i \leq W$ et qui maximise $\sum_{i=1}^{n} x_i v_i$.

Exercice 12. Proposer des algorithmes gloutons pour résoudre le pb du sac à dos ; sont-ils optimaux ?
(par valeurs décroissantes, v_i/w_i décroissants)

Programmation dynamique 13 :
① On considère les sous-pb $SD(i, w)$ réduit aux i premiers objets avec une capacité w. La solution qui nous intéresse est celle de $SD(n, W)$.
② Etant donné une variable i et une capacité w, les solutions optimales de $SD(i, w)$ sont soit :
- les solutions optimales du pb à $i-1$ variables avec capacité $w - w_i$, et alors on prend l'objet i ($x_i = 1$)

- les solutions optimales du problème à $i-1$ variables avec la même capacité w, et on ne prend pas l'objet ($x_i = 0$)

Si on note $T(i,w)$ la valeur optimale de $SD(i,w)$, alors

$$T(i,w) = \begin{cases} 0 \text{ si } i = 0 \text{ ou } w = 0 \\ \max\left(T(i-1,w),\ T(i-1, w - w_i) + v_i\right) \text{ sinon} \end{cases}$$

Remarque 14 : Le pb de décision associé au pb de sac à dos est NP-complet. Ici, l'algorithme résout le pb d'optimisation en $O(S*n)$, or la taille de l'instance est en $\log_2(S) + n$, notre algorithme n'est pas polynomial.

III - Autres applications

III. 1 - L'algorithme de Floyd-Warshall

Instance : Un graphe orienté $G = (S, A)$ sans circuit absorbant, et une fonction de poids $w : A \to \mathbb{Z}$

Pb : Déterminer les valeurs des plus courts chemins entre toutes paires de sommets de G.

Programmation dynamique 15 :

① On numérote les sommets de G : $S = \{1, \dots, n\}$.
On s'intéresse aux sous-problèmes $FW(i,j,k)$ qui correspond au plus court chemin de i à j ayant ses sommets intermédiaires dans $\{1, \dots, k\}$. Les problèmes qui nous intéressent sont $FW(i,j,n)$ $\forall i \neq j$.

② Pour résoudre $FW(i,j,k)$, on remarque que deux cas sont possibles :

- le plus court chemin de i à j ne comprend pas le sommet k, alors $FW(i,j,k) = FW(i,j,k-1)$
- Soit il comporte une unique fois le sommet k (plus signifierait la présence d'un circuit de poids nul supprimable). Il s'agit alors de la concaténation de 2 chemins : l'un de i à k, et l'autre de k à j, ne comportant chacun que des sommets intermédiaires dans $\{1, \dots, k\}$.

On en déduit la relation $FW(i,j,k) = \begin{cases} w(i,j) \text{ si } k = 0 \\ \min\left(F(i,j,k-1),\ F(i,k,k-1) + F(k,j,k-1)\right) \end{cases}$

Algo 16 :

```
def Floyd-Warshall(G) :
    W = matrice d'adjacence de G
    pour k allant de 1 à n:
        pour i allant de 1 à n:
            pour j allant de 1 à n:
                W[i,j] = min (W[i,j], W[i,k] + W[k,j])
```

Rq 17 : On écrase la matrice au fur et à mesure car l'étape k ne dépend que de l'étape $k-1$.

III. 2 - Distance de Levenshtein (d'édition)

déf 18 : La distance de Levenshtein correspond au nombre minimum d'opérations (suppression, modification ou ajout d'une lettre) pour transformer une chaîne de caractère en l'autre.

Pour $a \in \Sigma$, $i \in \mathbb{N}$, on définit

- $ins_{a,i} : \Sigma^* \to \Sigma^*$ (insertion de la lettre a en position i)
- $sub_{a,i} : \Sigma^* \to \Sigma^*$ (modification de la $i$ème lettre en a)
- $sup_i : \Sigma^* \to \Sigma^*$ (suppression de la $i$ème lettre)

$lev(w_1, w_2) = \min\{k \in \mathbb{N} \mid \exists f_1, \dots, f_k \in \{ins_{a,i}, sub_{a,i}, sup_i \mid a \in \Sigma, i \in \mathbb{N}\} : w_2 = f_k \circ \dots \circ f_2 \circ f_1(w_1)\}$

Développement 2 : Présentation et Correction d'un algorithme de programmation dynamique pour la distance d'édition

Ne rien écrire hors du cadre.