Programmation dynamique

Cours 02

1 Stratégies algorithmiques

1.1 Définitions et exemples

Pour rappel, en informatique, on distingue l'algorithme (conception théorique et indépendante des langages) du programme (conception pratique rattachée à un langage précis). Dans ce chapitre, on va traiter d'une stratégie algorithmique.

Définition 1 : stratégie algorithmique

Une stratégie algorithmique est une méthode de conception d'un algorithme.

Il faut distinguer les stratégies algorithmiques des méthodes de programmation, propres à l'implémentation d'un algorithme. Il existe de nombreuses stratégies algorithmiques, dont certaines déjà vues en cours : diviser-pour-régner et algorithme glouton.

Pour déterminer le maximum d'un tableau, on peut prendre une approche diviser-pour-régner.

```python
def max_rec(tab):
    if len(tab) == 1:
        return tab[0]
    n = len(tab)
    m1 = max_rec(tab[n//2:])
    m2 = max_rec(tab[:n//2])
    return max(m1,m2)
```

Le code ci-dessus est récursif, celui de droite ne l'est pas. Les deux programmes font toutefois appel à la même stratégie diviser-pour-régner : la stratégie algorithmique est la même, mais la technique de programmation est différente.

```python
def max_non_rec(tab):
    a_traiter = [(0,len(tab)-1)]
    max_act = tab[0]
    while a_traiter != []:
        a,b = a_traiter.pop()
        if a == b:
            max_act = max(max_act,tab[a])
        else:
            milieu = (a+b)//2
            a_traiter.append((a,milieu))
            a_traiter.append((milieu+1,b))
    return max_act
```

Pour rappel, les deux stratégies connues jusqu'ici sont les suivantes :

Définition 2 : diviser-pour-régner

Pour résoudre un problème, on le divise en sous-problèmes, qu'on résout chacun indépendamment, pour remonter à la solution du gros problème.

Définition 3 : stratégie gloutonne

Pour résoudre un problème d'optimisation, on construit une solution petit à petit en prenant à chaque étape ce qui est le mieux localement.

Il est important de noter que ces définitions ne sont pas des définitions formelles.

La programmation dynamique est une « généralisation » du diviser-pour-régner :

Définition 4 : programmation dynamique

Pour résoudre un problème, on le divise en sous-problèmes, qui ne sont pas forcément indépendants les uns des autres, pour remonter à la solution du gros problème.

2 Un exemple : le problème du rendu de monnaie

On considère un système monétaire, par exemple l'euro : il y a des pièces ou billets de 1, 2, 5, 10, 20, 50, 100 et 200 €. M. Sogé, banquier de son état, doit faire l'appoint pour rendre une valeur de 53€. Comment peut faire M. Sogé pour faire l'appoint en utilisant le moins d'unités possibles ?

Définition 5

Le problème du rendu de monnaie est le problème suivant :

Donnée : une liste finie d'entiers $C = [c_0, c_1, \dots, c_{|c|-1}]$ et un entier N ;

Question : trouver des entiers $(a_k)_{0 \leqslant k \leqslant |c|-1}$ tels que $N = \sum_{k=0}^{|C|-1} a_k c_k = N$, de sorte que $\sum_{k=0}^{|C|-1} a_k$ soit minimale.

Par souci de simplicité, on supposera toujours ici que $c_0 = 1$, donc que tout entier N dispose d'une décomposition.

2.1 Une première approche par algorithme glouton

L'algorithme glouton consiste ici à construire la solution en choisissant au fur et à mesure la « plus grosse pièce » qui puisse encore passer.

Données : C une liste d'entiers, N un entier
$res \leftarrow$ tableau rempli de 0 de longueur $|C|$
$r \leftarrow N$
tant que $r \neq 0$ **:**
  trouver $c_i \in C$ maximal tel que $c_i \leqslant r$
  $res[i] \leftarrow res[i] + 1, r \leftarrow r - c_i$
retourner res

L'algorithme ci-dessus est de complexité $O(|C| \times K)$, où K est le nombre de pièces utilisées dans la solution optimale cherchée ; l'algorithme à droite est de complexité $O(|C|)$, ou $O(\log(|C|) \times |C|)$ s'il faut trier C au préalable.

L'algorithme à gauche permet de trouver une solution gloutonne au problème du rendu de monnaie. Il est possible d'améliorer considérablement l'algorithme en calculant directement $res[i]$ à l'aide d'une division euclidienne.

Données : C **triée**, N un entier
$res \leftarrow$ tableau rempli de 0 de longueur $|C|$
$r \leftarrow N$
pour *i de |C|-1 à 0* **:**
  $res[i] \leftarrow r // c_i$
  $r \leftarrow r \bmod c_i$
retourner res

M. Lloyds, banquier britannique, observe les formidables performances de l'algorithme de M. Sogé : il veut l'utiliser lui aussi ! Au Royaume-Uni, le système monétaire est légèrement différent : il y a des pièces de 1, 2, 5, 10, 20, 25, 50, 100 et 200 £. Un jour, il doit rendre une somme de 41£ : patatras !

Propriété 6

Il existe des listes C telles que l'algorithme glouton ne donne pas la solution optimale du rendu de monnaie.

Démonstration.

L'algorithme glouton donne $41 = 25 + 10 + 5 + 1$; pourtant il y a la solution $41 = 20 + 20 + 1$ (qui est la solution optimale).

Plus précisément, certains systèmes monétaires (la plupart) font que l'algorithme glouton donne la solution optimale : un tel système est appelé canonique. Une question importante devient alors la canonicité d'un système monétaire. Il existe toute une théorie autour des sytèmes canoniques, dont on peut retenir les principes suivants :

— il n'existe pas de caractérisation mathématique évidente des systèmes canoniques qui soit indépendante de la taille du système ;
— il existe des algorithmes permettant de savoir si un système est canonique (Kozen-Zaks) ;
— il existe des caractérisations dans le cas de petits systèmes (jusqu'à 6 pièces).

2.2 Une seconde approche : mémoïsation et programmation dynamique

Pour résoudre le problème du rendu de monnaie dans tout système monétaire, l'idée consiste à construire une solution pour N à partir de solutions pour les entiers plus petits.

2.2.1 Approche top-down

Prenons un système non-canonique $C = [1, 4, 9]$. Pour déterminer une décomposition de N, l'idée est alors de supposer avoir déjà une décomposition de $N-1$, $N-4$ et $N-9$. On prend la plus petite de ces trois décompositions : on en déduit une décomposition de N.

La programmation dynamique prend ici la forme d'une simple récursivité. Un point crucial est toutefois que ce programme est **particulièrement inefficace** en temps. La raison est qu'on fait appel à `rendu_monnaie` de nombreuses fois sur des valeurs de N identiques.
Question. Dessiner l'arbre d'appels récursifs pour $C = [1, 4, 9]$ et $N = 12$.

```
def rendu_monnaie(C,N):
    if N == 0:
        return 0
    R = []
    for  in range(len(C)):
        if N >= c[i]:
            R.append(rendu_monnaie(C,N-c[i]))
    return min(R)+1
```

Pour éviter ces appels redondants, on met alors en place la **mémoïsation** : on crée un tableau pour stocker les valeurs calculées.

```
def rendu_monnaie(C,N):
    tab = [-1 for n in range(N+1)]
    tab[0] = 0
    def aux(k):
        if tab[k] != -1:
            return tab[k]
        R = []
        for c in C:
            if k-c >= 0:
                R.append(aux(k-c))
        tab[k] = min(R)+1
        return tab[k]
    return aux(N)
```

Ici, `tab` est ce qui nous permet de faire la mémoïsation. La fonction `aux`, définie à l'intérieur de notre fonction principale, est une fonction auxiliaire : elle prend en paramètre `k`, qui sera notre valeur intermédiaire dont on cherche une solution. Si `tab[k]` est déjà remplie, alors il n'y a pas besoin de la calculer ; sinon, on fait les appels récursifs nécessaires.
Pour déterminer la complexité de notre programme, on distingue les appels à `aux` qui remplissent une case de ceux où la case est déjà remplie :

- remplir la case : un tel appel a lieu une seule fois pour tout $k \in [\![0, N]\!]$, et est en $O(|C|)$;
- case déjà remplie : un tel appel a lieu au plus $|C|$ fois, et est en $O(1)$.

Au total, on atteint donc une complexité temporelle en $O(|C| \times N)$.

2.2.2 Approche bottom-up

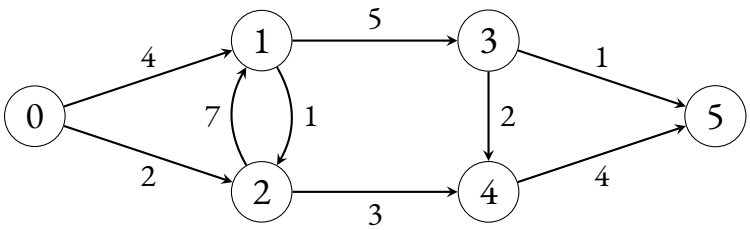
Dans l'approche précédente, on est parti de N, et on est descendu dans les appels récursifs. Mais quitte à remplir un grand tableau, on peut aussi le faire directement, en partant des petites valeurs faciles à calculer, et en montant dans le tableau : c'est l'approche bottom-up, par opposition à la top-down.
Les boucles imbriquées donnent une complexité temporelle en $O(|C| \times N)$.

```
def rendu_monnaie(C,N):
    tab = [-1 for n in range(N+1)]
    tab[0] = 0
    for k in range(1,N+1):
        m = N+1
        for c in C:
            if k-c >= 0 and tab[k-c]+1 < m:
                m = tab[k-c]+1
        tab[k] = m
    return tab[N]
```

On remarque qu'il n'y a pas de différence de complexité temporelle entre approches top-down et bottom-up.

3 Un autre exemple : distance dans un graphe orienté

Considérons un graphe orienté G : on le représentera ici sous forme d'une matrice d'adjacence `M_G`. Par exemple, `M_G[0][1] = 4`. S'il n'y a pas d'arête, on dira que le poids est infini.
En Python, on utilisera la notation `M_G[1][0]= float('inf')`, qui représente $+\infty$ (et est compatible avec la comparaison et les opérations arithmétiques).



Définition 7

Le problème de la distance entre deux sommets dans un graphe est le problème suivant :
Donnée : un graphe G et deux sommets s et t
Question : trouver un chemin de s à t de poids minimal.

3.1 Algorithme de Dijsktra

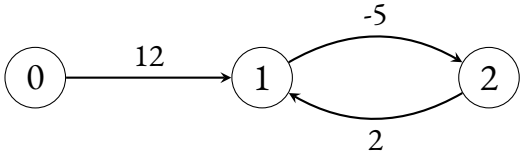
L'algorithme de Dijkstra est un algorithme efficace, qui fonctionne sous l'hypothèse que les arêtes sont de poids positifs. Il s'agit d'un algorithme glouton : pour choisir le nouveau sommet u à étudier, on prend le sommet le plus proche non encore traité.

La complexité de l'algorithme de Dijkstra dépend de l'implémentation des objets, et en particulier de l'étape d'extraction (qui permet d'obtenir u). En utilisant des tableaux, cette extraction se fait en $O(n)$ avec n le nombre de sommets ; au total, cet algorithme est en complexité temporelle $O(n^2)$, et complexité spatiale $O(n)$.

Données : M_G, s et t
$n \leftarrow$ nombre de sommets
$pred \leftarrow$ tableau rempli de 0 de longueur n
$dist \leftarrow$ tableau rempli de $+\infty$ de longueur n ; $dist[s] \leftarrow 0$
$S \leftarrow$ sommets du graphe
tant que *S n'est pas vide* **:**
  $u \leftarrow$ élément de S dont la $dist$ est minimale
  retirer u de S
  pour *tout sommet v* **:**
    $d \leftarrow dist[u] + M_G[u][v]$
    si $d < dist[v]$ **:**
      $dist[v] \leftarrow d$
      $pred[v] \leftarrow u$
retourner $dist[t]$ *et le chemin de s à t à partir de* $pred$

3.2 Algorithme de Bellman-Ford

L'un des problèmes de Dijkstra est qu'il ne fonctionne pas nécessairement correctement dans le cas où il y a un cycle de poids négatif : il ne détectera même pas forcément ce cycle.



L'algorithme de Bellman-Ford calcule un tableau $dist$ tel que $dist[u]$ contiendra, à la fin, la longueur du plus petit chemin de s à u. Le sous-problème étudié est le calcul des plus courts chemins depuis s vers tout sommet u en utilisant k arêtes, pour k allant de 0 (avant le premier passage de boucle) jusqu'à $n-1$. S'il n'y a pas de cycle négatif, le plus court chemin sera de longueur au plus $n-1$.

Données : M_G, s et t
$n \leftarrow$ nombre de sommets
$pred \leftarrow$ tableau rempli de 0 de longueur n
$dist \leftarrow$ tableau rempli de $+\infty$ de longueur n ; $dist[s] \leftarrow 0$
pour *k de 1 à $n-1$* **:**
  pour *tout couple de sommets (u, v)* **:**
    $d \leftarrow dist[u] + M_G[u][v]$
    si $d < dist[v]$ **:**
      $dist[v] \leftarrow d, pred[v] \leftarrow u$
retourner $dist[t]$ *et le chemin de s à t à partir de* $pred$

On peut modifier l'algorithme pour détecter les cycles négatifs : si, à la fin, il existe un couple (u, v) tel que $dist[v] > dist[u] + M_G[u][v]$, c'est que le graphe possède un cycle négatif.

L'algorithme de Bellman-Ford est de complexité temporelle $O(n^3)$ (et spatiale $O(n)$).

3.3 Algorithme de Floyd-Warshall

On s'intéresse désormais à une version légèrement différente du problème précédent : peut-on donner la distance de **tout** sommet vers tout sommet ?

Définition 8

Le problème de la distance entre tous sommets dans un graphe est le problème suivant :

Donnée : un graphe G

Question : calculer le tableau $dist$ tel que $dist[u][v]$ est la distance de u à v dans G.

Cette fois, le découpage en sous-problème est légèrement différent : il s'agira du calcul de la distance depuis tout sommet u vers tout sommet v, en se contraignant à utiliser comme sommets intermédiaires les sommets 0 à k.

L'algorithme de Floyd-Warshall est de complexité temporelle $O(n^3)$, et de complexité spatiale $O(n^2)$.

L'algorithme de Floyd-Warshall détecte les cycles négatifs : on peut vérifier, à la fin, s'il y a une valeur strictement négative sur la diagonale de $dist$.

On peut alléger ce code, en remarquant que les copies de $dist$ sont inutiles, et qu'on peut travailler dans une seule matrice qu'on met à jour. On remarque en effet que s'il n'y a pas de cycle négatif, il n'y a jamais, lors de la boucle k, de mise à jour d'aucun $dist^{(k)}[u][k]$ ou $dist^{(k)}[k][v]$, donc le calcul de d est correct même sans recopie de $dist$. S'il y a des cycles de poids négatifs, il y aura des problèmes de mise à jour, mais on détectera la présence de cycles négatifs.

Données : M_G
$n \leftarrow$ nombre de sommets
$dist^{(-1)} \leftarrow$ matrice $n \times n$ qui est une copie de M_G
pour k *de 0 à* $n-1$ **:**
  $dist^{(k)} \leftarrow$ copie de $dist^{(k-1)}$
  pour *tout couple de sommets* (u, v) **:**
    $d \leftarrow dist^{(k-1)}[u][k] + dist^{(k-1)}[k][v]$
    si $d < dist^{(k)}[u][v]$ **:**
      $dist^{(k)}[u][v] \leftarrow d$
retourner $dist^{(n-1)}$

3.4 Codes entiers

```
def extraire_min(l,d):
    ind = 0
    for i in range(len(l)):
        if d[l[i]] < d[l[ind]]:
            ind = i
    return l[ind],l[:ind]+l[ind+1:]
```

Une légère différence par rapport à l'algorithme donné est l'initialisation de `pred` : dans ce code, on a préféré dire que par défaut, un sommet est son propre prédécesseur.

```
def dijkstra(M_G,s,t) :
    n = len(M_G)
    pred = list(range(n))
    dist = n*[float('inf')]
    dist[s] = 0
    sommets = list(range(n))
    while sommets != []:
        u,sommets = extraire_min(sommets,dist)
        for i in range(n):
            d = dist[u]+M_G[u][i]
            if d < dist[i]:
                pred[i],dist[i] = u,d
    chemin,som = [t],t
    while som != s:
        som = pred[som]
        chemin = [som]+chemin
    return dist[t],chemin
```

```python
def bellman_ford(M_G,s,t):
    n = len(M_G)
    pred = list(range(n))
    dist = n*[float('inf')]
    dist[s] = 0
    for k in range(n):
        for u in range(n):
            for v in range(n):
                d = dist[u]+M_G[u][v]
                if d < dist[v]:
                    dist[v],pred[v] = d,u
    chemin,som = [t],t
    while som != s:
        som = pred[som]
        chemin = [som]+chemin
    return dist[t],chemin
```

```python
from copy import deepcopy

def floyd_warshall(M_G):
    n = len(M_G)
    dist = deepcopy(M_G)
    for k in range(n):
        for u in range(n):
            for v in range(n):
                d = dist[u][k] + dist[k][v]
                if d < dist[u][v]:
                    dist[u][v] = d
    return dist
```