Programmation dynamique

1 Introduction

Il existe différentes stratégies pour résoudre un problème, chacune avec ses limites : la méthode par *force brute*, (problème de coût), une *stratégie gloutonne* (stratégie pas toujours optimale) , une stratégie *diviser pour régner* (sous-problèmes nécessairement indépendants) . Ces stratégies ont été rencontrées en première année . Le problème du rendu de monnaie par exemple se résout à l'aide d'un algorithme glouton qui fournit une solution optimale avec certains systèmes de pièces. Les algorithmes dichotomiques et certains algorithmes de tri utilisent une stratégie diviser pour régner. On " divise" en réduisant un problème en sous-problèmes du même type qui ne se chevauchent pas, puis on " règne" en résolvant ces sous-problèmes. Il reste ensuite à combiner les solutions des sous-problèmes pour obtenir une solution au problème initial.

La *programmation dynamique* est envisagée si le problème présente la propriété de sous-structure optimale et que les chevauchements de nombreux sous-problèmes doivent être gérés.

Sous-structure optimale

Si on peut trouver une solution optimale à un problème en le découpant en sous-problèmes et en obtenant récursivement les solutions optimales des sous-problèmes, on dit que le problème vérifie la propriété de *sous-structure optimale*.

Chevauchement des sous-problèmes

Si des sous-problèmes ne sont pas indépendants et doivent être résolus plusieurs fois (ils sont répétés ou la récursivité les résout plusieurs fois au lieu de générer d'autres sous-problèmes) , on dit que les sous-problèmes se chevauchent.

La programmation dynamique est une technique de conception d'algorithmes qui permet de résoudre certains sous-problèmes de manière efficace.
Comme la stratégie de " diviser pour régner", elle est basée sur la résolution de sous-problèmes plus simples. Mais un problème est divisé en sous-problèmes qui ne sont pas forcément indépendants, qui peuvent en majorité se chevaucher entre eux. D'une certaine manière, on envisage au début tous les sous-problèmes comme dans une recherche exhaustive, mais on prend ses précautions pour ne pas avoir à les résoudre tous. C'est un point très important de ne résoudre qu'une seule fois chaque sous-problème afin de limiter le nombre de calculs et donc la complexité en temps.

On obtient alors une solution optimale au problème en combinant des solutions optimales aux sous-problèmes. Alors que dans une stratégie gloutonne, on avance en permanence en prenant à chaque étape la meilleure décision.

Ce type de résolution se prête bien à une écriture récursive plutôt qu'à une écriture itérative. Cependant , avec certains types de problèmes, l'écriture récursive peut être limitée par l'utilisation de la mémoire (taille de la pile d'appels récursifs) .
En général, cette méthode est appliquée à des problèmes d'optimisation. Un problème consiste à optimiser le coût d'une suite (finie) de prises de décisions. Cette suite de décisions correspond à un découpage du problème en sous-problèmes. On calcule des solutions optimales successives, comme pour un algorithme glouton, à des sous-problèmes qui peuvent se chevaucher et sont liés au problème par des relations de récurrence. Ensuite, c'est la combinaison de ces solutions qui produit une solution à un problème plus large.

Principe d'optimalité de Bellman

Définition : principe d'optimalité de Bellman La solution optimale à un problème d'optimisation combinatoire présente la propriété suivante : quel que soit l'état initial et la décision initiale prise, les décisions qui restent à prendre pour construire une solution optimale forment une solution optimale par rapport à l'état qui résulte de la première décision.

Définition : sous-structure optimale En informatique, un problème présente une sous-structure optimale si une solution optimale peut être construite à partir des solutions optimales à ses sous-problèmes.

Définition : programmation dynamique La programmation dynamique est une méthode de construction des solutions optimales d'un problème par combinaison des solutions optimales à des sous-problèmes. Pour cela, le problème considéré doit posséder une sous-structure optimale (définition précédente). Certaines combinaisons de solutions sont implicitement rejetées si elles appartiennent à un sous-ensemble qui n'est pas utile.
Le terme *programmation* est pris dans le sens *planification avec optimisation* : pour résoudre un problème, on planifie la résolution de sous-problèmes dont les solutions permettront d'obtenir la solution au problème. Le côté *dynamique* est dans l'étude des sous-problèmes et leur réutilisation.
L'expression *programmation dynamique* vient de l'anglais *dynamic programming* inventée par Richard Bellman au début des années 1950.

Il peut être nécessaire de mémoriser les solutions des sous-problèmes calculés à chaque étape afin de ne pas recommencer leur calcul. L'efficacité de la programmation dynamique tient en effet à ce que chaque sous-problème ne soit résolu qu'une seule fois. Le terme informatique est " mémoïsation" , de l'anglais " memoïzation". De manière générale, la complexité en temps est améliorée au détriment de la complexité en espace (comme très souvent, c'est une question de compromis). Deux options peuvent alors se présenter :

- approche de haut en bas, ou descendante, ou top-down en anglais, avec des appels récursifs jusqu'aux cas de base, et la mémoïsation qui apporte un gain de temps.
- Résolution de bas en haut, ou ascendante, ou bottom-up en anglais, avec itération des calculs à partir des plus petits sous-problèmes et stockage des résultats, qui entraîne un coût en espace, mais avec une possibilité de gain si on ne doit pas tout mémoriser.

Lorsqu'une fonction sauvegarde les résultats calculés, on parle de fonction à mémoire.

Deux exemples introductifs

Premier exemple : la suite de Fibonacci

Prenons l'exemple de la suite de Fibonacci. Les nombres f_n sont définis par $f_0 = 0$, $f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$ pour tout $n \geqslant 2$.
Une fonction récursive semble bienvenue pour calculer ces nombres .

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

Le calcul de f_n est ramené aux calculs de f_{n-1} et f_{n-2}. Le calcul de f_5 par exemple, demande le calcul de f_4 et f_3, le calcul de f_4 demande le calcul de f_3 et f_2, et le calcul de f_3 demande le calcul de f_2.
Autrement dit, la résolution d'un problème nécessite de résoudre plusieurs sous-problèmes qui ne sont pas indépendants, et de résoudre plusieurs fois certains sous-problèmes.
La fonction récursive ci-dessus est inutilisable pour calculer f_{100} par exemple. Le seul calcul de f_8 nécessite de calculer 13 fois f_2.

La complexité en temps est exponentielle.
Si $C(n)$ est le coût pour calculer f_n, alors $C(n) = C(n-1) + C(n-2) + O(1) \sim f_n$.

Appliquons le principe de mémoïsation afin de ne pas effectuer plusieurs fois les mêmes calculs. Pour cela , nous pouvons utiliser un dictionnaire. Ce dictionnaire peut être défini en dehors de la fonction, dans le corps de la fonction, ou en paramètre par défaut.

```
def fibo_mem(n,memo):
    if n in memo:
        return memo[n]
    if n==0 or n==1
        f=n
    else:
        f=fibo_mem(n-1,memo)+fibo_mem(n-2,memo)
    memo[n]=f
    return f
```

Lorsqu'on appelle la fonction la première fois, on met en argument un dictionnaire vide, et ensuite, on peut réutiliser le dictionnaire modifié.

```
memo={}
fibo_mem(5,memo)
memo
fibo_mem(45,memo)
```

Les nombres sont stockés dans un dictionnaire. Les clés de ce dictionnaire sont les entiers k et les valeurs associées les nombres f_k. La fonction renvoie la valeur de f_n si n est passé en paramètre.
Pour les appels récursifs, à la demande de f_k, soit la clé k est déjà dans le dictionnaire et on récupère la valeur, soit elle n'y est pas, et on demande les valeurs de f_{k-1} et f_{k-2}. Dès que l'on connaît un nouveau nombre, on le place dans le dictionnaire.
Les appels au dictionnaire memo sont en $O(1)$, d'où une complexité en $O(n)$ pour obtenir f_n.
Sur cet exemple, programmation dynamique = récursivité + mémoïsation. La programmation dynamique a été mise en oeuvre dans une approche descendante.

Dans une approche ascendante de la programmation dynamique, on supprime la récursivité en la remplaçant par une boucle. On commence par les sous-problèmes et on combine les solutions pour résoudre les problèmes les plus grands.
On utilise encore un dictionnaire pour stocker les résultats des calculs.

```
def fibo_asc(n):
    mem0={}
    for k in range(n+1):
        if k==0 or k==1:
            f=n
        else:
            f=memo[k-1]+memo[k-2]
        memo[k]=f
    return f
```

Remarque : les calculs sont effectués dans le même ordre que pour le programme récursif. Ici , on peut remplacer le dictionnaire par une liste.

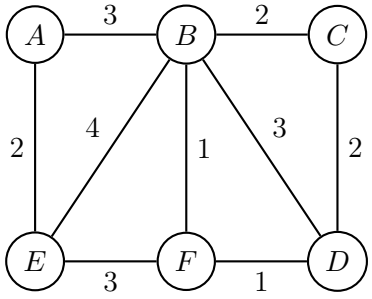
```
def fiboasc2(n):
    tab=[0,1]
    for i in range(2,n+1):
        tab.append(tab[i-1]+tab[i-2])
    return tab[n]
```

ou

```
def fibo_iter(n):
    u,v=0,1
    for i in range(n):
        u,v=v,u+v
    return u
```

Deuxième exemple : plus court chemin dans un graphe

On cherche le plus court chemin entre tous couples de sommets ou entre les sommets A et D du graphe représenté ci-dessous.



Une première approche consiste à examiner tous les chemins possibles (méthode par " force brute"). Avec quelques sommets, cette approche est envisageable, mais elle devient vite impraticable lorsque le nombre de sommets et d'arêtes augmente.

Une seconde approche consiste à appliquer un algorithme glouton : à partir de A, on rejoint à chaque étape le voisin le plus proche. On obtient alors ici le chemin $A-E-F-D$ pour un coût total de 6.

La programmation dynamique est entre les deux. Elle consiste à prendre en compte tous les cas, mais à n'en examiner pratiquement qu'un nombre restreint. Le principe de base est : si un chemin optimal de A à D passe par un sommet X, alors le chemin emprunté de A à X est optimal et le chemin emprunté de X à D est optimal.

Commençons par le début. On cherche un chemin sans intermédiaire (en une seule arête) partant de A
On a $A-B$, coût 3, et $A-E$ coût 2.
En rajoutant une arête, donc en cherchant les chemins partant de A et comportant au plus 2 arêtes , on obtient $A-B-C$ coût 5, $A-B-D$ coût 6, $A-B-F$ coût 4, $A-E-F$ coût 5 moins bien que $A-B-F$ donc on ne le garde pas, et $A-B-E$ coût 7 qui est moins bien que $A-E$ coût 3, donc on garde le chemin $A-E$ et on " supprime" l'arête $B-E$ et l'arête $E-F$
Enfin, en rajoutant encore une arête, donc en cherchant les chemins partant de A en au plus trois arêtes, on obtient que le coût du chemin $A-B-F-D$ est 5.
Cet algorithme est celui de Bellman-Ford, que l'on va étudier plus en détail après.
En cherchant le plus court chemin entre A et D, on a obtenu les plus courts chemins entre chacun des sommets et D. Donc pour résoudre un problème particulier, on a résolu un problème plus général , trouver tous les plus courts chemins entre un sommet quelconque et D, qui a été décomposé en sous-problèmes.

On résume la stratégie pour ce type de problème :

- ☐ Définir les sous-problèmes : déterminer les plus courts chemins entre deux sommets
- ☐ Évaluer les différentes possibilités, les choix à faire
- ☐ Établir des liens entre les solutions des sous-problèmes
- ☐ Utiliser dans une approche descendante la récursivité avec mémoïsation pour la résolution des sous-problèmes ou utiliser dans une approche ascendante une boucle avec construction d'une table pour stocker les résultats.

Algorithme de Bellman et Ford

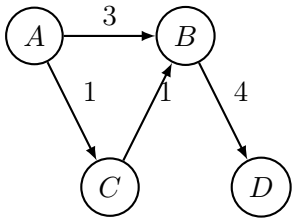
On considère un graphe $G=(S,A,w)$ un graphe orienté et pondéré ne possédant pas de cycles de poids négatif.
L'algorithme de Bellman et Ford suit le principe de la programmation dynamique. Les chemins les plus courts sont déterminés à partir d'un sommet donné s_0 jusqu'à tous les autres sommets.
Il s'agit de résoudre des sous-problèmes qui sont les calculs des distances entre le sommet s_0 et les autres sommets en au plus i arêtes, i allant de 0 à n, où n est le nombre de sommets du graphe. Ensuite , on combine les solutions.

Le principe est le suivant : toutes les distances initiales d'un sommet au sommet s_0 ont une valeur infinie. Sauf la distance de s_0 à lui-même qui vaut 0.
A chaque étape, on parcourt l'ensemble des arêtes, et on met à jour les distances :
pour chaque sommet v extrémité de fin d'une arête , si, en passant par un prédécesseur direct k du sommet v le chemin entre s_0 et v est plus court qu'à l'étape précédente, alors on passe par ce sommet k, et on met à jour la distance.

On recommence cette étape $n-1$ fois en suivant le même parcours des arêtes. A chaque étape, toutes les arêtes sont visitées au moins une fois. Si i représente **le nombre d'arêtes maximal autorisé pour atteindre un sommet** et $d_i(v)$ **la distance du sommet de départ** s_0 **au sommet** v en parcourant au maximum i arêtes, alors on a le résultat suivant :

$$d_i(v) = \begin{cases} 0 \text{ si } i = 0 \text{ et } v = s_0 \\ +\infty \text{ si } i = 0 \text{ et } v \neq s_0 \\ \min\left(d_{i-1}(v), \min_{arcs(k,v)}(d_{i-1}(k) + w(k,v))\right) \text{ sinon} \end{cases} .$$

Le graphe g est ici représenté par un dictionnaire des listes d'adjacence. Pour le graphe de la figure ci-dessous,

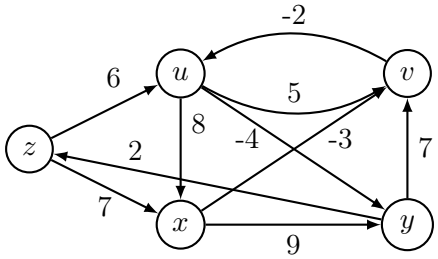


$g = \{'A' : [('C',\ 1),\ ('B',\ 3)],\ 'B' : [('D',\ 4)],\ 'C' : [('B',\ 1)],\ 'D' : [\ ]\}$.

```
def  bellmann_ford(g,s):
    n=len(g)
    d={k:float('inf') for k in g}
    d[s]=0
    for i in range(n-1):                    # on répète n-1 fois
        for k in g:                         # on teste tous les sommets
         for j in range(len(g[k]):      # et tous les voisins de ces sommets
             v,c=g[k][j]                # v sommet , c cout  de l'arete k-v
             if d[k]+c < d[v]:
                 d[v]=d[k]+c

    for k in g:                             # test de cycle négatif
        for j in range(len(g[k]):
            v,c=g[k][j]
            if d[v] > d[k]+c:
                return False
    return d
```

Tester le programme à la main pour le graphe suivant.



Et recommencer avec le même graphe, mais en modifiant le poids de l'arête yv à 4.

Si on note n le nombre de sommets et r le nombre d'arêtes du graphe, on peut montrer que la complexité de Bellman-Ford est en $O(nr)$.
Justification : initialisation en $O(n)$
Ensuite $n-1$ étapes, chacune en $O(r)$, donc $O(nr)$.

Algorithme de Floyd-Warshall

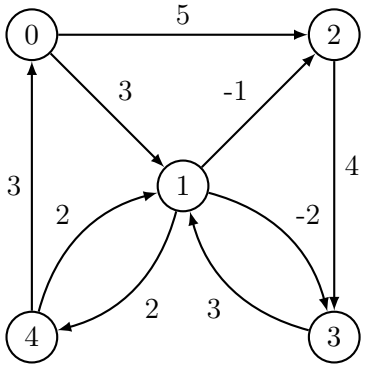
On considère un graphe orienté pondéré, avec des poids éventuellement négatifs, mais ce graphe ne doit contenir aucun cycle de poids négatif.
On suppose que les sommets de ce graphe sont nommés par des entiers entre 0 et $n-1$.
L'algorithme de Floyd-Warshall calcule, pour chaque paire de sommets du graphe, la distance entre les deux sommets. La distance est définie ici comme la longueur du plus court chemin, s'il en existe au moins un, entre les deux sommets.
Un chemin est une suite d'arcs, et sa longueur est la somme des poids des arcs.
Cet algorithme a été inventé par Bernard Roy en 1959 et publié en 1962 par Stephen Warshall puis Robert Floyd.

Principe d'optimalité : si A-K-F est le plus court chemin entre A et F, alors A-K est le plus court chemin entre A et K, et K-F est le plus court chemin entre K et F. Dans une séquence optimale, chaque sous-séquence est optimale.

A partir de la matrice d'adjacence représentant le graphe, on construit une matrice D_0 , la matrice des distances.
Un coefficient d'indice (i,j) égal à 0 dans la matrice d'adjacence signifie qu'il n'y a pas d'arc entre le sommet i et le sommet j. On pose alors $D_0(i,j)=\infty$.
On calcule ensuite les matrices D_k, après k itérations en utilisant des sommets intermédiaires dans $0,1,2,...,k-1$ qui donnent les longueurs des plus courts chemins passant par ces k sommets.
Ce sont les sous-problèmes à résoudre.
On note $c(s_1,s_2)$ le coût entre le sommet s_1 et le sommet s_2, soit le poids associé à l'arc (s_1,s_2) , et on écrit la relation de récurrence liant les sous-problèmes :
pour $0 \leqslant k \leqslant n$:

- si $k=0$, $D_k(s_1,s_2)=c(s_1,s_2)$ ($+\infty$ s'il n'y a pas d'arc entre s_1 et s_2)
- si $k \geqslant 1$, $D_k(s_1,s_2)=\min\limits_{v\in[\![0,1,\cdots,k-1]\!]}(D_{k-1}(s_1,s_2),D_{k-1}(s_1,v)+D_{k-1}(v,s_2))$

On prendra le graphe suivant comme exemple :



On utilise une approche ascendante pour programmer la résolution du problème.
Le graphe est ici représenté par un dictionnaire donnant la liste des couples (successeur , poids) de chaque sommet (liste vide si un sommet n'a pas de successeur).

```
G={0: [(1,3), (2,5)], 1: [(2,-1), (3,-2), (4,2)], 2: [(3,4)], 3: [(1,3)], 4: [(0,3), (1,2)] }
```

Calcul de la matrice des distances initiales :

```
from math import inf

def distances(g):
    n=len(g)
    m=[ [inf for j in range(n)] for i in range(n)]
    for s in g:
        m[s][s]=0
        for v,c in g[s]:
            m[s][v]=c
    return m
```

Avec D=distances(G), on obtient la matrice

```
D=[[0,3, 5, inf, inf], [inf, 0, -1, -2, 2], [ inf, inf, 0, 4, inf], [inf, 3, inf, 0, inf],
[3, 2, inf, inf, 0]]
```

On écrit le programme , puis on le teste avec la matrice D :

```
def floyd_warshall(D):
    n=len(D)
    for k in range(n):   # à la fin du passage pour k, D contient la matrice D_{k+1}
        for i in range(n):   # pour tous les sommets i
            for j in range(n):    # pour tous les sommets j
                if D[i][k]+D[k][j] < D[i][j]:
                    D[i][j]=D[i][k]+D[k][j]
    return D
```

```
>>> D=distances(G)
>>>floyd_warshall(D)
[[0,3,2,1,5], [5,0,-1,-2,2], [12,7,0,4,9], [8,3,2,0,5], [3,2,1,0,0]]
```

La complexité est en $O(n^3)$ comme pour le produit de deux matrices carrées.
La matrice initiale contient les longueurs des chemins n'empruntant qu'un arc. Pour $k = 0$, on considère les chemins passant par le sommet 0, pour $k = 1$, on considère les chemins passant par les sommets 0 et 1, et ainsi de suite.
L'algorithme est implanté en place , et la complexité spatiale est de l'ordre de n^2, l'espace nécessaire pour stocker la matrice des distances.
Pour prouver la correction, on peut montrer que $D_n \leqslant D_{k+1} \leqslant D_k$ pour tout $k \leqslant n-1$

```
def produit_matriciel(A,B):   #A et B sont deux matrices carrées de même taille
    n=len(A)
    C=[[0 for j in range(n)] for i in range(n)]  # création d'une matrice carrée
    for i in range(n):
        for j in range(n):   # calcul de AB[i][j]
            for k in range(n):
                C[i][j]=C[i][j]+A[i][k]*B[k][j]
    return C
```

Autres exemples

Les problèmes classiques sont :

- partition équilibrée d'un tableau d'entiers positifs
- ordonnancement de tâches pondérées
- plus longue sous-suite commune
- distance de Levenshtein
- rendu de monnaie
- sac à dos (voir prochain TP , sujet Mines 2025)