La programmation dynamique est un paradigme de conception d'algorithmes. Il permet de résoudre de nombreux problèmes, notamment d'optimisation.

L'idée est de ne pas recalculer plusieurs fois la même chose.

Ex: Fibonacci naïf → $O(\varphi^n)$ appels
avec un tableau → $O(n)$

I Exemple introductif : PLSC

$u = u_1 \cdots u_n$, $v = v_1 \cdots v_m$ deux mots de Σ^*.

On cherche une plus longue sous-séquence commune (PLSC) de u et v, ie. un mot $w = u_{i_1} \cdots u_{i_k} = v_{j_1} \cdots v_{j_k}$ $(1 \le i_1 < i_2 < \cdots \le n,\ 1 \le j_1 < j_2 < \cdots \le m)$ de longueur maximale.

① Sous-structure optimale

Prop : Soit $w = w_1 \cdots w_k$ une PLSC de u et v.

- si $u_n = v_m = \alpha$ alors $w_k = \alpha$ et $w_1 \cdots w_{k-1}$ est une PLSC de $u' = u_1 \cdots u_{n-1}$ et $v' = v_1 \cdots v_{m-1}$
- si $u_n \neq v_m$ alors
  - si $w_k \neq u_n$, w est une PLSC de u' et v
  - si $w_k \neq v_m$, ——— u et v'

② Formulation récursive

Cl° :
- Si $u = \varepsilon$ ou $v = \varepsilon$, on renvoie ε.
- Si $u_n = v_m = \alpha$, on cherche w' une PLSC de u' et v' et on renvoie $w'\alpha$.
- Sinon, on cherche une PLSC de u' et v et une de u et v' et on renvoie la plus longue des 2.

③ Calcul de la longueur d'une PLSC

On note $l[i,j]$ la longueur d'une PLSC de $u_1 \cdots u_i$ et $v_1 \cdots v_j$ $(0 \le i \le n,\ 0 \le j \le m)$.

On a alors :

$$l[i,j] = \begin{cases} 0 \text{ si } i=0 \text{ ou } j=0 \\ l[i-1,j-1]+1 \text{ si } i,j>0 \text{ et } u_i = v_j \\ \max(l[i-1,j], l[i,j-1]) \text{ sinon} \end{cases}$$

Dépendances :

Complexité : $O(nm)$

Pour calculer $l[n,m]$ on remplit la matrice ligne par ligne.

```
l[0..n, 0..m] initialisé à 0
pour i = 1 à n faire
    pour j = 1 à m faire
        si u_i = v_j alors
            l[i,j] ← l[i-1,j-1] + 1
        sinon
            l[i,j] ← max(l[i-1,j], l[i,j-1])
renvoyer l[n,m]
```

III Principes et compléments

1/ Principe

* trouver, dans une structure optimale, des sous-structures optimales → diviser en sous-problèmes
* ne calculer qu'une fois chaque problème → approche ascendante (≠ récursivité)

Conditions requises

* sous-structures optimales
* **chevauchement des sous-problèmes** (sinon = diviser pour régner)

Difficultés éventuelles

* trouver la sous-structure optimale
* trouver le bon ordonnancement pour les calculs

2/ Retrouver la solution

Notre algorithme ne renvoie que la longueur, pour retrouver une PLSC on peut créer un tableau $f[1..n, 1..m]$ que l'on remplit avec des ↖, ↑ ou ← en fonction de la valeur qui a servi à calculer $f[i,j]$.

On retrouve alors une PLSC avec :

$PLSC(i,j) =$
* $*$ si $i = 0$ ou $j = 0$, ε
* $*$ si $f[i,j] =$ ↖, $PLSC(i-1, j-1)\, u_i$
* $*$ si $f[i,j] =$ ↑, $PLSC(i-1, j)$
* $*$ si $f[i,j] =$ ←, $PLSC(i, j-1)$

$\Rightarrow O(k)$

Si on n'a pas maintenu le tableau on peut remonter "à la main" avec la même complexité.

3/ Optimisation mémoire

Souvent, on peut optimiser l'algorithme pour **réduire** sa complexité en espace. Notamment si on ne cherche que la valeur de l'extremum et non la structure optimale (cf distance d'édition).

Ex pour PLSC :

On ne garde que deux lignes du tableau et on remplace les $f[\square, \Delta]$ par des $f[\square \bmod 2, \Delta]$. On peut même ne garder qu'une ligne et une case.

4/ Rescencement, Memoization

Technique qui consiste à maintenir une table des valeurs déjà calculées et à garder une approche descendante.

Ex pour PLSC :

$\ell[0..n, 0..m]$ initialisé à $\perp$

LONGUEUR (i,j) =
- si $\ell[i,j] = \perp$ alors
  - si $i=0$ ou $j=0$, alors $\ell[i,j] \leftarrow 0$
  - si $u_i = v_j$ alors $\ell[i,j] \leftarrow 1 + $ LONGUEUR $(i-1, j-1)$
  - sinon $\ell[i,j] \leftarrow \max($LONGUEUR$(i-1,j),$ ——$(i,j-1))$
- renvoyer $\ell[i,j]$

⊕ * formulation simple
* peut être efficace si on ne sait pas quels sous-problèmes calculer
* peut être utilisé d'un appel sur l'autre

⊖ * généralement moins efficace qu'une approche ascendante bien faite

Rq : Il existe des techniques de mémoïzation automatique. Ex en Maple : "option remember".

III Exemples

1/ Plus court chemin dans un DAG

$G=(S,A)$ graphe orienté sans cycle (DAG) pondéré.
$s \in S$, on cherche un chemin $s \rightsquigarrow v$ de poids min pour tout $v \in S$

Si $s \rightsquigarrow v_1 \to v_2$ optimal, alors $s \rightsquigarrow v_1$ optimal.

$\Rightarrow d(s, v_2) = \min d(s, v_1)$

Algorithme :
* Trier S dans l'ordre topologique $\rightsquigarrow S = \{s_1, s_2, \dots s_n\}$. avec $s = s_{i_0}$
* $d[1..n]$ initialisé à $+\infty$
* pour $i = 1$ à n tq $i \neq i_0$ faire
  $d[s_i] \leftarrow \min_{(v,s_i) \in A} (d[v] + w(v, s_i))$

$\Rightarrow O(S)$

Rq : Cet exemple est fondamental car tous les algorithmes de programmation dynamique s'y ramènent !

2/ Distance d'édition [OV1]

On cherche le nb minimal d'opérations pour passer de x à $y \in \Sigma^*$ parmi :
- ajouter une lettre
- supprimer ——
- modifier ——

Si $x = x'\alpha$ et $y = y'\beta$ alors :
$d(x,y) = \min(d(x',y') + \delta_{\alpha \neq \beta}, d(x',y), d(x,y'))$
$\Rightarrow$ on obtient un algorithme en $O(|x| \cdot |y|)$

3/ Plus court chemin dans un graphe [OV2]

$d^{(k+1)}(x,y) = \min(d^{(k)}(x,y), d^{(k)}(x,k+1) + d(k+1,y))$
$\rightarrow$ c'est l'algorithme de Floyd-Warshall, $O(S^3)$

Catalogue de pbs :

* parenthesage pour chaîne de matrice [Cor, Papa]
* Fibonacci (x)
* arbres binaires de recherche optimaux [Eroi ?]
* PLSC [Cor]
* distance d'édit° [Papa]
* Floyd-Warshall [Cor]
* CYK [Carton]
* sac à dos [Papa]
* voyageur de commerce [Papa]

et encore plus ds le Kleinberg-Tardos ... ?

Bonus : memoization automatique en Caml

[Wikipedia, Memoization]

Version 1

```
let memo f =
  let Hbl = Hashtbl.create 100 in
  let g x =
    try Hashtbl.find Hbl x
    with Not_found -> let y = f x in
                      Hashtbl.add Hbl x y; y
  in g
```

Ex : `let fibo1 x = if x<2 then x else fibo1(n-1)+fibo1(n-2)`

```
let fibo2 = (memo fibo1)
fibo2(50); fibo2(70) ...
```

Pb : On ne capture pas les appels récursifs !
⇒ il faut passer le corps de la fonction en argument à memo.
C'est une grosse astuce.

Puis, par exemple :

```
let fibo3 = memo2 (fun f x -> if x<2 then x
                              else f(x-1)+f(x-2))
```

f est un argument !!

Version 2

```
let memo2 f =
  let Hbl = H.create 100 in
  let rec g x =
    try H.find Hbl x
    with Not_found -> let y = (f g x) in
      H.add Hbl x y; y
  in (f g)
```

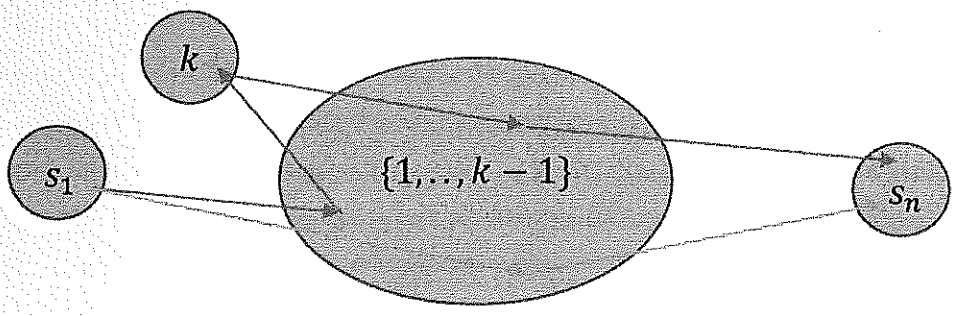
Algorithme de Floyd-Warshall

On se donne un graphe fini orienté dont les arêtes ont un poids entier. On suppose qu'il n'y a pas de cycles de poids négatifs dans le graphe et on cherche à déterminer les chemins de poids minimaux reliant chaque couple de sommets du graphe.

Structure d'un chemin de longueur minimale

Si $p = (s_1, s_2, .., s_l)$ est un chemin, on appelle sommet intermédiaire un des sommets $s_i, 1 < i < l$. On suppose pour simplifier que l'ensemble des sommets soit $\{1, .., n\}$. Supposons que p soit un chemin de poids minimal reliant i à j dont tous les sommets intermédiaires sont dans $\{1, .., k\}$. On peut toujours supposer que p est sans cycle. On a alors deux cas possibles :

- Si k n'est pas un sommet intermédiaire de p. Alors p est aussi un chemin de poids minimal reliant i à j dont tous les sommets intermédiaires sont dans $\{1, .., k-1\}$.
- Si k est un sommet intermédiaire de p. Alors on peut décomposer $p = (s_1, .., s_n)$ en $p_1 = (s_1, .., s_i = k)$ et $p_2 = (s_i = k, .., s_n)$. Alors p_1 et p_2 sont des chemins optimaux reliant respectivement i à k et k à j et n'ayant que des sommets intermédiaires dans $\{1, .., k-1\}$.



Expression de la formule de récurrence

Si on note $d_{i,j}^{(k)}$ le poids minimal d'un chemin reliant i à j et n'ayant que des sommets intermédiaires dans $\{1, .., k\}$, on a alors la formule de calcul récursive :

$$d_{i,j}^{(0)} = w_{i,j}.$$

$$d_{i,j}^{(k)} = \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right), k \geq 1.$$

Ordre de calcul des $d_{ij}^{(k)}$

Pour calculer $d_{i,j}^{(k)}$, on a uniquement besoin des $d_{i,j}^{(k-1)}$. On peut donc ordonner (i, j, k) par k croissant.

Algorithme

Algorithme naïf : on stocke une matrice $(n+1) \times n \times n$ contenant les $d_{i,j}^{(k)}$ et on la remplit en initialisant $d_{i,j}^{(0)}$ avec les poids du graphe. En fait, on peut faire mieux, et avoir une complexité $O(n^2)$ en espace :

```
Entrée : matrice des poids W du graphe.
Retourne : matrice D telle que D[i,j] est le poids minimal
d'un chemin de i à j.
FLOYD-WARSHALL(W) :
1.   D = W
2.   Pour k = 1 à n
3.     Pour i = 1 à n
4.       Pour j = 1 à n
5.         D[i,j] = min(D[i,j], D[i,k]+D[k,j])
6.   Retourne D
```

On n'a pas besoin de garder tous les $d_{i,j}^{(k)}$, car on a, à chaque assignement de $D[i, j]$,

$$d_{i,j}^{(n)} \leq D[i, j] \leq d_{i,j}^{(k)}:$$

en effet, la suite $d_{i,j}^{(k)}$ est décroissante et la fonction de récurrence laisse $d_{i,j}^{(n)}$ fixe par inégalité triangulaire.

On a donc une complexité en temps $O(n^3)$ et en espace $O(n^2)$.

Comparaison avec l'algorithme de Dijkstra

Par rapport à effectuer l'algorithme de Dijkstra sur tous les sommets du graphe, les différences sont les suivantes :

- La complexité ne dépend pas du nombre d'arêtes : l'algorithme de Floyd-Warshall est donc à recommander dans le cas de graphes très denses.

- L'algorithme de Dijkstra est plus difficile à implémenter, et nécessite dans sa déclinaison la plus efficace un tas de Fibonacci.
- La complexité asymptotique avec tas de Fibonacci de Dijsktra est $O(m + n\log(n))$ qui est bien meilleure dans le cas de graphes peu denses.
- L'algorithme de Floyd-Warshall autorise des poids négatifs, et interdit seulement les cycles de poids négatif.

Distance d'édition

On cherche à définir une notion de distance entre mots, par exemple pour un correcteur orthographique, qui compare un mot et cherche dans son dictionnaire pour des mots qui lui sont proches.

Une distance naturelle entre deux mots, par exemple POIRE et POMME est de trouver un alignement des lettres des deux mots, i.e. un moyen d'écrire les deux mots l'un au-dessus de l'autre, tel que le nombre de lettres qui diffèrent entre les deux soit minimal.

```
_POIRE POIRE
POM_ME POMME
```

Exemple : le premier alignement comporte 5 différences et le deuxième n'en comporte que 2.

On appelle cette distance *distance d'édition,* car elle correspond en fait au nombre minimal d'opérations parmi

- Insérer une lettre
- Supprimer une lettre
- Remplacer une lettre par une autre

à réaliser pour passer du premier mot au deuxième.

On a un algorithme de programmation dynamique qui, étant donné deux mots $x = (x_1, .., x_m)$ et $y = (y_1, .., y_n)$, permet de trouver la distance d'édition entre x et y.

Structure des sous-problèmes

Notation : si x est un mot on note $x[i, j] = (x_i, .., x_j)$ le facteur de x composé des lettres de la $i$ème à la $j$ème.

On se fixe deux mots $x = x[1, m]$ et $y = y[1, n]$. Si on a un alignement qui réalise la distance d'édition entre $x[1, i]$ et $y[1, j]$, alors on a plusieurs cas :

- On a remplacé x_i par y_j et le mot $x[1, i-1]$ est aligné de manière optimale à $y[1, j-1]$. Deux sous-cas sont à traiter : $x_i = y_j$ ou $x_i \neq y_j$.
- x_i est effacé, i.e. x_i est aligné à un blanc. Alors l'alignement restant de $x[1, i-1]$ avec $y[1, j]$ est optimal.
- On a inséré une lettre après x_i, i.e. y_j est aligné à un blanc. Alors l'alignement restant de $x[1, i]$ avec $y[1, j-1]$ est optimal.

Expression de la formule de récurrence

Notons $E(i,j)$ la distance d'édition de $x[1,i]$ à $y[1,j]$, on a alors, par ce qui a été dit avant :

Cas de base :

$$E(i,0) = i.$$
$$E(0,j) = j.$$

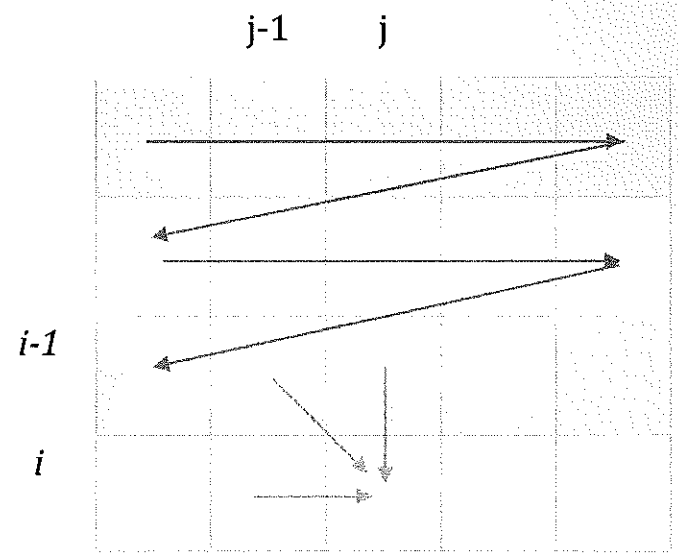
Formule de récurrence :

$$E(i,j) = \min\big(1 + E(i-1,j), 1 + E(i,j-1), diff(i,j) + E(i-1,j-1)\big),$$

où $diff(i,j) = \begin{cases} 0 \text{ si } x[i] = y[j] \\ 1 \text{ sinon} \end{cases}$.

Ordre sur les sous-problèmes

Pour calculer $E(i,j)$, on a besoin de $E(i,j-1)$,$E(i-1,j)$ et $E(i,j)$.



On peut donc calculer les $E(i,j)$ en i et j croissants.

Algorithme

A partir de là, l'algorithme est naturel :

```
Distance_edition(x,y) :
   1. m = taille(x)
   2. n = taille(y)
   3. E = créer_tableau((m,n))
   4. Pour i = 0 à n
   5.    E(i,0) = i
```

```
6. Pour j = 0 à m
7.    E(0,j) = j
8. Pour i = 1 à m
9.    Pour j = 1 à n
10.       Si x[i] = y[j]
11.         E(i,j) = min(E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1))
12.       Sinon
13.        E(i,j) = min(E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1)
14.Retourne E(m,n)
```