Notes de cours – CM3

1 Programmation dynamique

1.1 Un premier exemple

Dans ce chapitre, nous allons aborder une technique algorithmique classique : la programmation dynamique. Grossièrement, il s'agit d'accélérer les calculs en stockant en mémoire les solutions de sous-problèmes, ce qui évite de les recalculer plusieurs fois. On sacrifie donc la complexité en espace pour améliorer la complexité temporelle.

Prenons l'exemple de l'algorithme suivant, qui calcule le n-ème terme de la suite de Fibonacci.

Algorithme 1 : $f(n)$

Entrées : Un entier n
Sorties : Le n-ème terme de la suite de Fibonacci

```
1 si n ⩽ 1 alors
2 |  retourner n
3 sinon
4 |  retourner f(n − 1) + f(n − 2)
```

Si on essaye de calculer $f(5)$, on se rend compte qu'on doit calculer $f(4)$ et $f(3)$. Mais pour calculer $f(4)$, on calcule $f(3)$ et $f(2)$. Autrement dit, l'ensemble des calculs peut être représenté par l'arbre suivant :

TODO DESSIN

On peut se rendre compte que ces arbres ont une taille exponentielle en n. En particulier, sur cet exemple, on se rend compte qu'on recalcule trois fois $f(2)$ et deux fois $f(3)$. On va donc plutôt calculer les $f(i)$ pour $i = 0, \ldots, n$ et les stocker pour ne pas avoir à les recalculer. On obtient donc l'algorithme suivant : On obtient alors une complexité linéaire !

Algorithme 2 : $f(n)$

Entrées : Un entier n
Sorties : Le n-ème terme de la suite de Fibonacci

```
1 T ← tableau de taille n + 1
2 T[1] ← 1
3 pour i = 2, . . . , n faire
4 |  T[i] ← T[i − 1] + T[i − 2]
5 retourner T[n]
```

1.2 Multiplication de matrices

L'opération de multiplication de matrices rectangulaires prend deux matrices de taille $m \times n$ et $n \times p$ et renvoie une matrice de taille $m \times p$ en utilisant (en gros) mnp opérations. Cette opération est associative, c'est-à-dire que $(A \times B) \times C = A \times (B \times C)$ mais pas commutative $AB \neq BA$ en général. Ceci signifie que lorsqu'on doit calculer un produit $A_1 \times \cdots \times A_n$, on peut effectuer les multiplication dans l'ordre qu'on souhaite, tant qu'on n'échange pas les termes.

Imaginons qu'on veuille calculer $ABCD$ où A est de taille 50×20, B de taille 20×1, C de taille 1×10 et D de taille 10×100. On pourrait parenthéser comme $A((BC)D)$ ce qui coûterait $20 \times 1 \times 10 + 20 \times 10 \times 100 + 50 \times 20 \times 100 = 120200$. Mais peut-on faire mieux ?

Si on applique l'algorithme glouton consistant à commencer par l'opération la moins coûteuse, on ferait $(A(BC))D$, ce qui coûte $20 \times 1 \times 10 + 50 \times 20 \times 10 + 50 \times 10 \times 100 = 60200$.

Mais en fait on peut faire mieux : $(AB)(CD)$ coûte $50 \times 20 \times 1 + 1 \times 10 \times 100 + 50 \times 1 \times 100 = 7000$. On voit donc que l'algorithme glouton ne fonctionne pas, et il nous faut trouver une autre méthode.

On peut naturellement représenter les produits possibles par des arbres binaires (TODO DESSIN). Chaque sous-arbre représente une manière d'effectuer un produit de la forme $A_i \cdots A_j$. La remarque fondamentale ici est que chaque sous-arbre d'un arbre optimal doit aussi être optimal (autrement on pourrait remplacer le sous-arbre par un arbre moins coûteux, et le coût total diminuerait). Ainsi, on voit apparaître un sous-problème naturel : comment calculer les produits $A_i \cdots A_j$? Notons $C_{i,j}$ le coût de calcul de ce produit.

On a facilement que $C_{i,i} = 0$. Pour calculer $C_{i,j}$, on peut se rendre compte que le calcul de $A_i \cdots A_j$ nécessite de choisir k entre i et j, de calculer les matrices $A_i \cdots A_k$ et $A_{k+1} \cdots A_j$, puis de les multiplier. Ainsi, si A_i a taille $m_{i-1} \times m_i$, on a

$$C_{i,j} = \min_{i \leqslant k < j} (C_{i,k} + C_{k+1,j} + m_{i-1} m_k m_j)$$

On se retrouve à nouveau avec une relation de récurrence, comme pour Fibonacci. On remarque que pour calculer $C_{i,j}$ on n'utilise que des valeurs $C_{k,\ell}$ avec $\ell - k < j - i$. On peut donc calculer les valeurs de $C_{i,j}$ par ordre de $j - i$ croissant et les stocker dans un tableau. Ceci donne l'algorithme suivant :

Algorithme 3 : Multiplication de matrices

Entrées : Les tailles de n matrices
Sorties : Le coût optimal du calcul de leur produit

```
pour i = 1, ..., n faire
    C[i][i] ← 0
pour s = 1, ..., n − 1 faire
    pour i = 1, ..., n − s faire
        j ← i + s
        C[i][j] ← min_{i⩽k<j} (C_{i,k} + C_{k+1,j} + m_{i−1} m_k m_j)
retourner C[1][n]
```

Concernant la complexité, on voit qu'on stocke $O(n^2)$ valeurs. Pour calculer chacune de ces valeurs, on calcule un min en temps linéaire. La complexité finale est donc $O(n^3)$.

Le message à retenir de ce paragraphe (et même du chapitre) est que la programmation dynamique est utilisée lorsqu'on peut identifier des sous-problèmes tels que les sous-structures d'une solution globale sont des solutions optimales des sous-problèmes. On peut alors résoudre petit à petit les sous-problèmes et stocker leurs solutions jusqu'à résoudre le problème entier.

1.3 Algorithme de Floyd-Warshall

On se donne un graphe $G = (S, A)$ où les arêtes sont pondérées de sorte qu'il n'y ait pas de cycle négatif. Le but est de calculer les $|S|^2$ distances entre chaque paire de sommets. L'hypothèse interdisant les cycles négatifs est nécessaire pour que ces distances soient bien définies.

Pour rappel, l'algorithme de Dijkstra calcule la distance entre deux sommets donnés en $O(|A| + |S| \log |S|)$ si les poids sont positifs. Et l'algorithme de Bellman-Ford calcule la distance entre un sommet et tous les autres en $O(|S||A|)$. On pourrait donc appliquer $O(|S|^2)$ fois l'algorithme de Dijkstra (si les poids sont positifs) ou bien $O(|S|)$ fois l'algorithme de Bellman-Ford, ce qui conduit à une complexité de $O(|S|^2|A|)$.

On va faire mieux avec l'algorithme de Floyd-Warshall, basé sur la programmation dynamique. La question à se poser est donc : quels vont être les sous-problèmes à considérer ? Si on cherche à restreindre l'ensemble des paires de sommets qu'on considère, on ne va pas faire mieux que les $|S|$

applications de Bellman-Ford. Il faut donc restreindre autre chose, et on n'a pas beaucoup plus de choix que de restreindre l'autre entrée du problème, à savoir G.

Ainsi, pour calculer les distances dans G, on va calculer les distances dans des sous-graphes de G où on ne considère que certains sommets. Définissons $d(i, j, k)$ comme la distance entre les sommets i et j où on ne peut utiliser comme sommets intermédiaires ceux de numéro inférieur à k. En particulier, quand $k = n - 1$, on obtient bien les distances dans G.

On se rend facilement compte que $d(i, j, 0)$ est le poids de l'arête ij si elle existe, et $+\infty$ sinon. Il s'agit maintenant de trouver une relation de récurrence sur $d(i, j, k)$. TODO DESSIN

Pour aller de i à j en utilisant les sommets d'indice au plus k, on peut soit choisir de ne pas utiliser k, et la distance est alors $d(i, j, k-1)$, soit on utilise k, et la distance est $d(i, k, k-1) + d(k, j, k-1)$. On a donc

$$d(i, j, k) = \min\{d(i, j, k-1), d(i, k, k-1) + d(k, j, k-1)\}$$

En calculant ces valeurs pour $k = 0, \ldots, n-1$, on obtient donc l'algorithme suivant :

Algorithme 4 : Algorithme de Floyd-Warshall

Entrées : Un graphe pondéré $G = (|S|, |A|, w)$
Sorties : La matrice des distances entre chaque paire de points

```
1 pour i, j = 1, ..., n faire
2 |  D[i][j][0] ← ∞
3 |  D[i][i][0] ← 0
4 pour ij ∈ A faire
5 |  D[i][j][0] ← w(ij)
6 pour k = 1, ..., n faire
7 |  pour i, j = 1, ..., n faire
8 |  |  D[i][j][k] ← min(D[i][j][k − 1], D[i][k][k − 1] + D[k][j][k − 1])
9 retourner D[][][n]
```

La complexité temporelle de cet algorithme est donc $O(n^3)$. On stocke de plus un tableau $n \times n \times n$, donc on utilise aussi $O(n^3)$ en mémoire.

En revanche, on aurait pu être plus économe en mémoire : en effet, on peut se rendre compte que le calcul de $D[i][j][k]$ n'utilise que des éléments de $D[][][k-1]$. Il est donc inutile de conserver $D[][][p]$ en mémoire si $p < k - 1$. Avec cette amélioration, on obtient l'algorithme suivant, dont la complexité temporelle est inchangée, mais dont la complexité spatiale est maintenant $O(n^2)$.

Algorithme 5 : Algorithme de Floyd-Warshall – version optimisée

Entrées : Un graphe pondéré $G = (|S|, |A|, w)$
Sorties : La matrice des distances entre chaque paire de points

```
1 pour i, j = 1, ..., n faire
2 |  D[i][j] ← ∞
3 |  D[i][i] ← 0
4 pour ij ∈ A faire
5 |  D[i][j] ← w(ij)
6 pour k = 1, ..., n faire
7 |  D' ← D
8 |  pour i, j = 1, ..., n faire
9 |  |  D[i][j] ← min(D'[i][j], D'[i][k] + D'[k][j])
10 retourner D
```

On obtient ainsi la matrice des distances entre chaque paire de points. Mais on a aussi envie de pouvoir retrouver les plus courts chemins. Pour ce faire, on va retenir pour chaque paire (i, j) et chaque

entier k, le sommet $p_{i,j,k}$ qui précède j dans un plus court chemin de i à j n'utilisant que des sommets d'index au plus k. La relation de récurrence vérifiée par $p_{i,j}$ est alors

$$p_{i,j,k} = \begin{cases} p_{i,j,k-1} & \text{si } d(i,j,k-1) < d(i,k,k-1) + d(k,j,k-1) \\ p_{k,j,k-1} & \text{si } d(i,j,k-1) \geqslant d(i,k,k-1) + d(k,j,k-1) \end{cases}$$

avec la condition initiale $p_{i,j,0} = i$ si $i = j$ ou $ij \in A$. À nouveau, le calcul de $p_{i,j,k}$ ne dépend que de valeurs $p_{a,b,k-1}$, on peut donc seulement retenir ces valeurs, et on obtient l'algorithme :

Algorithme 6 : Algorithme de Floyd-Warshall – calcul des chemins

Entrées : Un graphe pondéré $G = (|S|, |A|, w)$
Sorties : La matrice des distances entre chaque paire de points

```
1  pour i, j = 1, ..., n faire
2      D[i][j] ← ∞
3      D[i][i] ← 0
4      P[i][i] ← i
5  pour ij ∈ A faire
6      D[i][j] ← w(ij)
7      P[i][j] ← i
8  pour k = 1, ..., n faire
9      D' ← D
10     P' ← P
11     pour i, j = 1, ..., n faire
12         D[i][j] ← min(D'[i][j], D'[i][k] + D'[k][j])
13         si D'[i][j] ⩾ D'[i][k] + D'[k][j] alors
14             P[i][j] ← P'[k][j]
15 retourner D, P
```

Pour récupérer le chemin de i à j, il suffit donc de calculer $P[i][j]$, puis $P[i][P[i][j]]$, ... jusqu'à tomber sur i.

1.4 En résumé

— Décomposer en sous-problèmes (préfixes, infixes, sous-arbres, sous-graphes)
— Écrire une relation de récurrence. Si ça ne marche pas, décomposer autrement.
— Calculer les valeurs dans le bon ordre
— Est-ce qu'on doit tout retenir à tout moment ?