Chapitre 3
Programmation dynamique (partie 2)

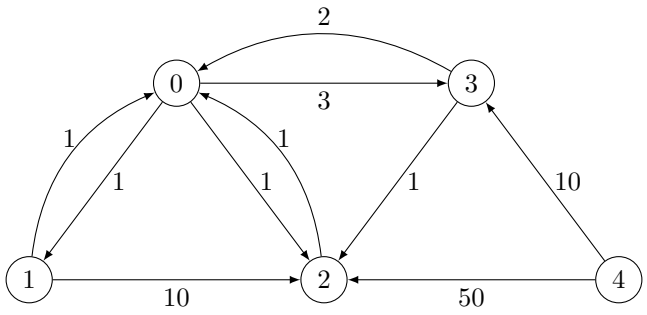
Dans le précédent chapitre, nous avons vu un problème de programmation dynamique que nous avons résolu par récursivité, après avoir établi une formule de récurrence. Comme les appels récursifs apparaissaient plusieurs fois, nous avons du implémenter une méthode par mémoïsation pour ne pas refaire des calculs de façon redondante. Dans ce chapitre, nous allons voir une autre méthode de programmation dynamique qui n'utilisera ni récursivité ni mémoïsation.

Table des matières

1 Étude d'un exemple : plus court chemin dans un graphe

1.1 Résolution par programmation dynamique

Supposons que l'on ait un graphe orienté et valué :



Alors, on peut former la matrice d'adjacence de ce graphe avec la convention suivante : s'il existe une arc qui part du sommet i et qui va vers le sommet j, le coefficient $m_{i,j}$ sera égal à la valeur de cet arc, sinon on met $+\infty$ (par convention). Dans l'exemple ci-dessus, on obtient alors la matrice :

$$M = \begin{pmatrix} +\infty & 1 & 1 & 3 & +\infty \\ 1 & +\infty & 10 & +\infty & +\infty \\ 1 & +\infty & +\infty & +\infty & +\infty \\ 2 & +\infty & 1 & +\infty & +\infty \\ +\infty & +\infty & 50 & 10 & +\infty \end{pmatrix}$$

Étant donnés deux sommets i et j on cherche la longueur du plus court chemin qui mène de i à j, on a donc bien un problème d'optimisation. Rappelons qu'un chemin de i vers j est de la forme $i \to s_1 \to s_2 \to \ldots \to s_r \to j$ où les s_k sont des sommets intermédiaires et tels qu'il existe une arête de i vers s_1, une arête de s_1 vers s_2 etc,de s_r vers j. La longueur du chemin est donc

$$m_{i,s_1} + m_{s_1,s_2} + \ldots + m_{s_{r-1},s_r} + m_{s_r,j} = m_{i,s_1} + \sum_{i=1}^{r-1} m_{s_i,s_{i+1}} + m_{s_r,j}$$

et on cherche à minimiser cette longueur parmi tous les chemins possibles partant de i et allant vers j (s'il n'en existe pas, alors on dira que la longueur du chemin vaut, par convention, $+\infty$).

Conservons les principes de programmation dynamique déjà vus, procédons en deux étapes :

- D'abord chercher la longueur du plus court chemin
- Puis chercher le plus court chemin en lui même.

On va stocker le résultat, du plus court chemin, dans une matrice [1] L : pour tout $(i,j) \in [\![0\,;n-1]\!]^2$, $\ell_{i,j}$ vaut la longueur du plus court chemin de i à j s'il existe, $+\infty$ sinon [2].

Pour cela, on va aborder ce problème en découpant ce problème en des sous-problèmes que l'on va résoudre de proche en proche. On va chercher le plus court chemin de i à j en passant par des sommets intermédiaires dont les numéros sont inférieurs ou égales à k. C'est-à-dire que l'on va considérer des chemins de la forme

$$i \to s_1 \to s_2 \to \ldots \to s_r \to j \quad \text{avec} \quad (s_1, s_2, \ldots, s_r) \in [\![0\,;k]\!]^r$$

Notez que i et j ne sont pas forcément dans $[\![0\,;k]\!]$, ce sont seulement les sommets intermédiaires qui doivent être entre $[\![0\,;k]\!]$. On note alors $\ell_{i,j}^{(k)}$ la longueur du plus court chemin parmi ces tels chemins [3]. Et $L^{(k)} = (\ell_{i,j}^{(k)})_{\substack{0\leqslant i\leqslant n-1 \\ 0\leqslant j\leqslant n-1}}$. La matrice $L^{(k)}$ contient alors la longueur du plus court chemin de i à j dont les sommets intermédiaires dont les sommets sont inférieurs à k pour tous les i et j. On remarque que l'on cherche $L = L^{(n-1)}$, en effet chercher le plus court chemin entre i et j sans restriction sur les sommets intermédiaires, revient à chercher le plus court chemin entre i et j avec des sommets intermédiaires dans $[\![0\,;n-1]\!]$.

1. Contrairement à l'usage en mathématiques, nous allons numéroter nos lignes et colonnes à partir de 0, ainsi pour une matrice $L \in \mathscr{M}_n(\mathbb{R})$, on a $L = (\ell_{i,j})_{\substack{0\leqslant i\leqslant n-1 \\ 0\leqslant j\leqslant n-1}}$

2. S'il existe $(i,j) \in [\![0\,;n-1]\!]^2$ tel qu'il n'existe pas de chemin de i vers j, on dit que le graphe n'est pas connexe.

3. Attention, le terme puissance (k) ne désigne pas une puissance, mais c'est un exposant pour désigner que l'on autorise seulement les sommets entre 0 et k comme sommets intermédiaires.

Cherchons comment on passe de $L^{(k)}$ à $L^{(k+1)}$. Considérons deux sommets i et j, on cherche le plus court chemin entre i et j passant par des points intermédiaires qui sont entre 0 et $k+1$. Considérons un tel plus court chemin. Il y a deux cas :

- Si ce plus court chemin ne passe par par $k+1$, alors $\ell_{i,j}^{(k+1)} = \ell_{i,j}^{(k)}$.
- Si ce plus court chemin passe par $k+1$, alors, il est de la forme

$$i \to \ldots \to k+1 \to \ldots \to j$$

Mais alors, nécessairement, le chemin entre i et $k+1$ est un chemin optimal qui n'a aucun intérêt de repasser par $k+1$, autrement dit le chemin de i à $k+1$ est un chemin optimal passant par des sommets entre 0 et k et a donc pour longueur $\ell_{i,k+1}^{(k)}$ de même pour le chemin entre $k+1$ et j, ainsi le chemin optimal a pour longueur :

$$\ell_{i,j}^{(k+1)} = \ell_{i,k+1}^{(k)} + \ell_{k+1,j}^{(k)}$$

Le hic, c'est que comme on ignore quel est le plus court chemin, on ignore, en particulier, s'il passe par $k+1$ ou non. Cependant, comme on cherche le plus court chemin, il suffit de prendre le plus court entre les deux possibilités. Ainsi :

$$\ell_{i,j}^{(k+1)} = \min(\ell_{i,k+1}^{(k)} + \ell_{k+1,j}^{(k)}, \ell_{i,j}^{(k+1)})$$

Fort de ce principe, on peut donc écrire l'algorithme de (Roy)-Floyd-Warshall qui va boucler sur k, i et j et qui va à chaque fois minimiser les coefficients te la matrice :

```
def Floyd(M:list)->list:
    """Étant donné un graphe donné par une matrice M, renvoie une matrice L
    où L[i,j] est la longueur du plus court chemin entre i et j
    """
    L=[[M[i][j] for j in range(n)] for i in range(n)] # copie de M, L=M.copy() me marche pas
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if L[i][k] + L[k][j] < L[i][j]:#Si passer par k est plus court
                    L[i][j] = L[i][k] + L[k][j]#Alors on actualise la distance entre i et j
    return L
```

1.2 Reconstitution du plus court chemin

Avoir la longueur du plus court chemin est certes bien pratique, mais il ne faut pas perdre de vue, que l'on cherche surtout le chemin en lui même. Pour cela, on va créer une autre matrice S telle que $S[i][j]$ indique l'arrêt suivant par lequel il faut passer pour aller de i vers j. Autrement dit, le plus court chemin entre i et j sera de la forme :

$$i \to S[i][j] \to \ldots \to j$$

Ainsi, `S[i][j]` contient le successeur de `i` dans le trajet qui mène de `i` vers `j`. On initialise cette matrice avec initialement que des[4] -1. De plus, si on a un arc de `i` à `j`, alors, on peut initialiser avec `S[i][j]=j` :

```
S = [[0 for j in range(n)] for i in range(n)] #Matrice des successeurs
for i in range(n):
    for j in range(n):
        if M[i][j]<np.inf:#s'il existe une arrête de i vers j
            S[i][j]=j#trajet direct de i vers j
```

4. Initialiser à 0 semblerait plus intuitif, mais cela pourrait donner l'illusion que l'on pense que pour aller du sommet `i` à `j` il faut forcément passer par le sommet 0 (la station Abessesses en l'occurrence)

Et à chaque fois, qu'on trouve un k qui minimise le chemin, on actualise S :

```
def Floyd(M:list)->list:
    """Étant donné un graphe donné par une matrice M, renvoie une matrice L
    où L[i][j] est la longueur du plus court chemin entre i et j
    """
    L=[[M[i][j] for j in range(n)] for i in range(n)] # copie de M, L=M.copy() me marche pas
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if L[i][k] + L[k][j] < L[i][j]:#Si passer par k est plus court
                    L[i][j] = L[i][k] + L[k][j]#Alors on actualise la distance entre i et j
                    S[i][j] = S[i][k]#On indique que pour aller de i à j il faut passer par le premier sommet
#intermédiaires du plus court chemin entre i et k
return L
```

Ainsi, si on veut aller d'un sommet `dep` vers un sommet `arr`, il faut partir de `dep`, puis passer par un sommet `sommet`=P[`dep`,`arr`], puis faire comme si on partait de `sommet` et qu'on voulait aller à `arr`, ainsi le prochain sommet visité sera `sommet2`=P[`sommet`,`arr`], on continue ainsi tant qu'on est pas arrivé au sommet final qui est `arr` :

```
def Itineraire(dep,arr):
    chemin=[dep]
    sommet=dep
    while sommet!=arr:
        sommet=Di[int(S[D[sommet],D[arr]])]#Attention S est une matrice, il lui faut donc un
        #numéro de ligne qui correspond à sommet: D[sommet], un numéro de colonne qui correspond à
        #arr: D[arr], mais S est une matrice Python qui renvoie le numéro du prochain sommet par
        #le quel il faut passer, mais une matrice Python ne contient que des nombres à virgules
        #(des flottants) on le convertie en entier grâce à int, mais on ne veut pas le numéro
        #de l'arrêt de métro, on veut son nom, on utilise donc le dictionnaire inverse, pour
        #que la variable sommet contienne le nom du sommet et non son numéro
        chemin.append(sommet)#On le rajoute à la liste chemin
    return chemin
```

1.3 Terminaison/Complexité/Correction

- La terminaison est évidente : des boucles `for` avec des `range(n)` sont nécessairement finies. [5]
- Pour (i,j,k) fixé, le `if` ne fait que comparer deux nombres et actualise un coefficients dans deux matrices (ce qui se fait en temps constant). Il y a ainsi trois boucles `for` imbriquées les unes dans les autres, ce qui donne une complexité en $\mathcal{O}(n^3)$.
- On observe qu'à la fin de l'étape k, la matrice N vaut la matrice L^k. En effet, dans la boucle `for`, on actualise la matrice N de la même manière que l'on calcule les coefficients de $L^{(k+1)}$ en fonction de ceux de $L^{(k)}$. Pour le dire savement, $N = L^{(k)}$ est un invariant de boucle. Ainsi, à la fin de la dernière itération sur `k`, $N = L^{(n-1)}$ comme voulue. Encore une fois, les variants de boucle permettent de démontrer la correction de l'algorithme [6].

1.4 Quelles sont les différences entre l'algorithme de (Roy)-Floyd-Warshall et celui de Dijkstra ?

L'algorithme de Dijkstra donne les plus courts chemins entre un sommet `source` spécifié et n'importe quelle autre sommet `but` Avec l'algorithme de Roy-Floyd-Warshall, on a plus d'informations car on a les plus courts chemin entre n'importe quel sommet `source` et n'importe quel sommet `but`.

L'algorithme de RFW a une complexité en n^3 si n est le nombre de sommets, Dijkstra a une bien meilleure complexité [7] (on fait une boucle sur tous les sommets, mais à chaque fois on recherche le sommet non traité le plus proche)

On peut aussi remarquer que l'implémentation de l'algorithme de Roy-Floyd-Warshall est plus simple.

5. On peut noter que l'on peut faire des boucles `for` infinie, mais il vaut vraiment le vouloir, par exemple :
```
L=[3,5]
for k in L:
    L.append(8)
```
6. On remarque de plus, que la façon dont a trouvé l'algorithme de Roy-Warshall-Floyd permet directement de démontrer la correction, en effet, on actualise `N` de façon à ce que directement $N = L^{(k)}$ soit un variant de boucle.
7. Cependant, l'étude de la complexité de l'algorithme de Dijkstra est plus compliquée, nous admettrons donc ce point.

2 Principe de bas en haut versus haut en bas (mémoïsation)

Ici, on a résolu un problème avec de la programmation dynamique : on a un problème, trouver la matrice L qu'on a résolu en résolvant des sous-problèmes (en l'occurrence trouver $L^{(k)}$). Cependant, ici on a abordé le problème sans que l'on ait besoin de travailler par récursivité. En effet, on a trouvé une façon d'organiser nos calculs tels que l'on dépende uniquement de calculs déjà effectués. En effet, on trouve les coefficients de $L^{(k+1)}$ à partir de $L^{(k)}$, on progresse ainsi jusqu'à arriver à $L = L^{(n-1)}$. On dit qu'on a fait de la programmation dynamique de bas en haut.

La programmation dynamique par mémoïsation s'appelle la méthode de haut en bas, en effet on demande directement la valeur d'en haut en fonction de valeurs plus basses et la récurisvité fait le job. Le fait que les sous-problèmes soient utilisés plusieurs fois fait que la mémoïsation permet de ne pas faire des calculs plusieurs fois et permet donc que le programme achève son calcul dans un délai raisonnable.

D'une certaine manière, la méthode bas en haut évite la récursivité en indiquant ce qu'on doit calculer en premier.

Savoir si la mémoïsation est une meilleure méthode que celle de bas en haut est une question à laquelle il est difficile de répondre en toute généralité. Pour certains problèmes, cela dépend plus des goûts de chacun. À noter qu'il est possible d'écrire une version de cet algorithme de haut en bas (avec récursivité et mémoïsation donc).

3 Récursivité/mémoïsation/bas en haut/algorithme glouton quels différences ?

3.1 Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même, pour que cela ait une chance de fonctionner il faut que l'algorithme renvoie directement une valeur pour certains paramètres sans appel récursif. Il faut, de plus, que les appels récursifs puissent eux-même se résoudre avec d'autres appels récursifs qui a leur tout dépendront uniquement par des appels de la fonction avec ces paramètrès calculables :

```
def AlgoRec(P):
    """Squelette d'un algo récursif"""
    if conditions sur P:# Étape d'initialisation, on peut mettre plusieurs conditions et renvoyer une valeur
        #dans chacun des cas
        return un truc explicite sans appel de AlgoRec
    Valeur=un calcul #fait en fonction de AlgoRec(P') pour un ou plusieurs P'
    return Valeur
```

On peut faire de la récursivité sans faire de la programmation dynamique (par exemple le tri fusion), on peut aussi faire de la programmation dynamique sans récursivité (toute méthode de bas en haut comme l'algorithme de Roy-Floyd-Warshall).

3.2 Mémoïsation

La mémoïsation est utile lorsque lors d'un algorithme récursif, la fonction va s'appeler elle-même plusieurs fois avec les mêmes paramètres. Alors, les calculs vont être effectués plusieurs fois à l'identique par Python. C'est comme si vous calculiez 58786*874561 trente fois d'affilée et qu'à chaque fois vous jetiez le résultat et que vous deviez recommencer. À la place, il suffit de mémoriser. Rappelez-vous qu'il ne s'agit pas seulement de réduire la durée d'un calcul, il s'agit parfois de passer d'un calcul qui prendrait quasiment une éternité à un calcul qui va être fait en quelques secondes. La contrepartie [8] c'est que l'on va perdre un peu en stockage dans la mémoire.

```
D={}#Dictionnaire qui va enregistrer les calculs déjà faits (ne surtout pas le mettre dans la fonction
#au risque d'effacer le dictionnaire à chaque appel de la fonction).
def AlgoMémoisation(P):
    """Squelette d'un algo récursif avec mémoïsation"""
    if P not in D:#on va calculer uniquement si on est en présence d'un calcul qui n'a jamais été fait
        if conditions sur P:#Initialisation, on peut mettre plusieurs conditions et renvoyer
            #une valeur dans chacun des cas
            D[P]=un truc explicite sans appel de AlgoRec
        D[P]=calcul#Étape d'hérédité le calcul est en fonction de AlgoMémoisation(P') pour un ou plusieurs P'
    return D[P]#Dans les trois cas (valeur déjà calculé, initialisation, récursivité) on renvoie
    #la valeur stockée dans le dictionnaire
```

8. Il y en a toujours une.

À noter qu'on peut faire de la mémoïsation sans forcément faire de la programmation dynamique (cf. la programmation de la suite de Fibonacci lors du TP2). On peut faire de la programmation dynamique sans nécessairement faire de la mémoïsation (par la méthode de bas en haut comme l'algorithme de Roy-Floyd-Warshall).

3.2.1 Bas en haut

```
def AlgoBasEnHaut(Param):
    M = une liste (ou une liste de listes) destinée à stocker les valeurs déjà calculés
    for ... in ... :
        #On rentre dans M les valeurs initiales
    for ... in ... :
        #On stocke dans M des valeurs calculés à l'aide de valeurs déjà stockés par M
        #Il faut donc réfléchir à l'ordre des calculs.
```

3.3 Algorithme Glouton/programmation dynamique

Rappelons ce qu'est **algorithme glouton** : c'est un algorithme qui se résout par une succession d'étapes où à chaque étape :

— On a fait un choix qui a réduit au maximum la taille du problème.
— Et que ce choix est définitif.

Ainsi, le rendu de pièces de monnaies vu en PCSI est un algorithme glouton. À chaque étape, on cherche la plus grande pièce pour réduire au maximum le montant qui reste à rembourser. Et ce choix est définitif, on ne revient donc jamais dessus (quitte à passer à côté d'une solution plus optimale : dans ce cas, qui prenne moins de pièces).
On pourrait penser qu'un algorithme glouton est un algorithme de programmation dynamique mais ce n'est pas le cas. En effet, à chaque étape d'un algorithme qlouton, on optimise l'étape, mais on ne change pas les étapes précédentes. Alors qu'en programmation dynamique, on résout de façon plus globale. Cependant la programmation dynamique et un algorithme glouton ont en commun d'avoir un programme qui se résout étapes par étapes [9].

9. Par opposition à la programmation dynamique, où tant que les calculs ne sont pas tous menés on a aucune idée des premières étapes de la solution recherchée, un algorithme glouton dessine la solution au fur et à mesure sans jamais dévier du chemin déjà pris, pourrait-on appeler un algorithme glouton un algorithme statique ?