Exercices d'algorithmes d'approximation. Programmation dynamique.

Absil Romain

18 mai 2011

Table des matières

1 Introduction.

La programmation dynamique est une technique classique dans le design d'algorithmes d'optimisation. Le principe de base est de construire la solution optimale d'un problème à partir de solutions optimales de sous-problèmes, le plus souvent stockées dans des tableaux, matrices ou autres structures de données multidimensionnelles.

Quand elle est applicable et utilisée judicieusement, cette méthode offre souvent de meilleurs résultats qu'une approche naïve. En effet, lors de la

décomposition d'un problème en sous-problèmes, on peut être amené à résoudre plusieurs fois un même sous-problème. La clé de ce principe est donc de résoudre chaque sous-problème exactement une fois.

Cette technique va donc réduire considérablement le temps de calcul, plus particulièrement quand le nombre de "sous-problèmes répétés" est arbitrairement grand.

La première fois introduite par Bellmann [2], cette méthode est à présent largement utilisé en informatique, notamment en

- bioinformatique : algorithmes de Needleman-Wunsch et de Smith-Waterman d'alignement maximal de deux séquences de nucléotides,
- théorie des graphes : algorithme de Floyd de calcul de tous les plus courts chemins,
- infographie : algorithme de Boore d'évaluation des B-Splines,
- analyse numérique : calcul des moindres carrés récursifs, etc.

2 Exemple : la suite de Fibonacci.

Le calcul de la suite de Fibonacci $F(n)$, définie ci-dessous, illustre un bel exemple de l'utilité de la programmation dynamique.

$$F(n) = \begin{cases} 1 & \text{si } n \in \{1, 2\}, \\ F(n-1) + F(n-2) & \text{sinon.} \end{cases}$$

Il est bien connu qu'implémenter une récursion naïve pour calculer la suite de Fibonacci est peu efficace, dans la mesure où on calcule plusieurs fois un même résultat, comme illustré par la Figure 1. On peut même montrer que cette implémentation a une complexité en $\mathcal{O}(\varphi^n)$.

Supposons à présent que l'on dispose d'une structure de donnée M capable d'associer à chaque valeur k une valeur v, telle qu'un tableau. On notera $M(k)$ la valeur v associée à k dans M. On peut dès lors modifier l'algorithme naïf afin d'utiliser cette structure et la mettre à jour pour diminuer drastiquement la complexité des calculs. On obtient donc l'algorithme de programmation dynamique ci-dessous. Cet algorithme a une complexité en temps dans le pire des cas linéaire.

Notons que la programmation dynamique est une méthode exacte, et n'a donc à priori pas de lien direct avec les algorithmes d'approximation.

Toutefois, on peut concevoir des algorithmes d'approximation en arrondissant les données *en entrée* du problème (c'est d'ailleurs ce qui est généra-

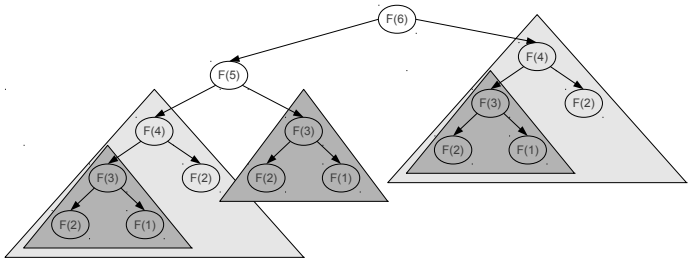


FIGURE 1 – Illustration de la redondance des calculs lors de l'évaluation de la suite de Fibonacci.

Algorithm 1 Algorithme Fib(n)

1: $M(1) = M(2) = 1$
2: **retourner** $F(n)$

3: **Fonction** F(n)
4: **si** $M(n)$ n'est pas défini
5: $\quad M(n) = F(n-1) + F(n-2)$
6: **retourner** $M(n)$

lement effectué dans le cas de tels algorithmes).

Cette technique se distingue donc des autres méthodes d'approximation car un arrondi est effectué avant l'exécution d'un quelconque algorithme, sur son entrée, et non pas après son exécution, sur sa sortie.

3 Éléments de programmation dynamique.

Bien que l'exemple ci-dessus illustre l'utilité de la programmation dynamique pour améliorer l'efficacité de la résolution de certains problèmes, il est intéressant de caractériser la structure des problèmes pour lesquels cette technique est toute indiquée.

Dans le cas présent, on s'intéressera plus particulièrement aux problèmes d'optimisation, qui peuvent être résolus rapidement grâce à des algorithmes d'approximation.

Concrètement, ces problèmes doivent avoir les propriétés suivantes :
– Sous-structure d'optimalité,
– Recouvrement des sous-problèmes,
– Memoisation [1].

3.1 Sous-structure d'optimalité.

Souvent, la première étape dans la résolution d'un problème d'optimisation est la caractérisation de la structure d'une solution optimale.

Plus particulièrement, un problème a une *sous-structure optimale* si une solution optimale du problème « contient » des solutions optimales de sous-problèmes.

Quand un problème partage cette propriété, il peut être intéressant de le résoudre par programmation dynamique. Dans ce cas, on construira la solution optimale d'un problème à partir de solutions optimales de sous-problèmes, par une approche *bottom up*.

L'exemple ci-dessous illustre la sous-structure d'optimalité du problème de plus court chemin dans un graphe non-orienté. Toutefois, il faut parfois se montrer prudent avec cette propriété, et ne pas assumer qu'elle s'applique là où elle n'est pas préservée, comme illustré ci-après.

Considérons les problèmes suivants :
– Recherche du plus court chemin élémentaire entre deux sommets,
– Recherche du plus long chemin élémentaire entre deux sommets.

Le problème de recherche de plus court chemin exhibe une sous-structure d'optimalité. Soit un plus court chemin p non trivial de u à v, et w un sommet sur ce chemin. On peut décomposer le chemin $u \overset{p}{\rightsquigarrow} v$ en sous-chemins $u \overset{p_1}{\rightsquigarrow} w$ et $w \overset{p_2}{\rightsquigarrow} v$.

Clairement, le nombre d'arêtes dans p est égal à la somme du nombre d'arêtes dans p_1 et dans p_2. De plus, si p est optimal (*i.e.,* de longueur minimum), alors p_1 et p_2 sont optimaux [2].

On pourrait donc assumer que le problème de recherche de plus long chemin entre deux sommets u et v a une sous-structure d'optimalité. Toutefois, ce n'est pas le cas, comme illustré par l'exemple de la Figure 2.

1. Ceci n'est pas une faute d'orthographe. Le terme est bien mémoisation et non mémorisation. Mémoisation vient de *memo*, cette technique consistant à enregistrer une valeur destinée à être consultée ultérieurement.
2. La preuve de cette propriété simple est laissée en exercice.

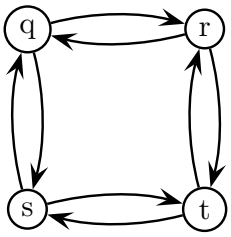


FIGURE 2 – Contre-exemple de la sous-structure d'optimalité pour le problème de plus long chemin.

Sur cet exemple, on voit que $q \longrightarrow r \longrightarrow t$ est un plus long chemin, de q à t. Toutefois, ni $q \longrightarrow r$, ni $r \longrightarrow t$ ne sont des plus longs chemins de q à r ou de r à t.

En effet, les chemins $q \longrightarrow s \longrightarrow t \longrightarrow r$ et $r \longrightarrow q \longrightarrow t \longrightarrow t$ sont des chemins de q à r et de r à t de longueur supérieurs.

Par ailleurs, il est impossible de construire un plus long chemin à partir de solutions de sous-problèmes. En effet, si l'on construit un plus long chemin de q à t à partir des chemins $q \longrightarrow s \longrightarrow t \longrightarrow r$ et $r \longrightarrow q \longrightarrow t \longrightarrow t$, on obtient le chemin $q \longrightarrow s \longrightarrow t \longrightarrow r \longrightarrow q \longrightarrow t \longrightarrow t$, qui n'est pas élémentaire.

Le problème du plus long plus court chemin ne partage donc pas la propriété de sous-structure d'optimalité. Rivest *et al.* [1] affirment même qu'aucun algorithme de programmation dynamique n'a jamais été trouvé pour ce problème.

3.2 Recouvrement des sous-problèmes.

Une seconde caractéristique que les problèmes devraient partager afin que la programmation dynamique soit applicable et efficace est la propriété de *recouvrement des sous-problèmes.*

Intuitivement, cela signifie d'un algorithme récursif pour un problème va résoudre de nombreuses fois des sous-problèmes identiques, plutôt que de générer des nouveaux sous-problèmes.

Typiquement, les algorithmes de programmation dynamique tirent avantage de cette propriété en résolvant chaque problème une unique fois et en stockant sa solution dans une table où elle pourra être consultée ou mise à jour rapidement.

On remarque que l'évaluation de la suite de Fibonacci illustrée à la Figure 1 partage cette propriété.

En effet, on remarque que la hauteur de l'arbre binaire d'évaluation de $F(n)$ est en $\mathcal{O}(n)$. Dès lors, cet arbre a un nombre k de nœuds avec k en $\mathcal{O}(2^n)$. Or, l'espace des sous-problèmes a une taille valant exactement n (il y a exactement n problèmes distincts à résoudre), on a donc un facteur de recouvrement de l'ordre de $2^n/n$.

En revanche, l'algorithme d'Euclide de calcul du pgcd illustré ci-dessous ne partage pas la propriété de recouvrement des sous-problèmes.

Algorithm 2 Algorithme pgcd(a,b)

```
1: si b = 0
2:    retourner a
3: sinon
4:    retourner pgcd(b, a mod b)
```

En effet, tous les sous-problèmes générés par la procédure récursive sont indépendants, et l'on n'est jamais amené à résoudre plusieurs fois un même problème. Utiliser la programmation dynamique dans ce cas est donc inutile.

En considérant l'exemple plus général de la résolution d'un problème linéaire en nombre entiers, la taille de l'espace des sous-problèmes générés par une procédure de Branch and Bound ne peut pas être bornée (car ces problèmes sont tous indépendants). Dès lors, la programmation dynamique n'est pas applicable pour cet exemple.

3.3 Memoisation.

Ce principe a déjà été préalablement débattu. Étant donné qu'un algorithme de programmation dynamique résout chaque sous-problème du problème initial exactement une fois, il est nécessaire de conserver les solutions de ces sous-problèmes, dans le cas où ils seraient amenés à être résolus à nouveau.

L'idée de base est de *memoiser* l'algorithme récursif naturel (mais inefficace). D'ordinaire, on maintient une table avec les solutions des sous-problèmes, la structure de contrôle permettant de remplir cette table étant l'algorithme récursif.

Selon le problème, on peut utiliser des tableaux (unidimensionnels ou non), des arbres ou des tables de hachage dans le but de stocker et/ou mettre à jour les valeurs de solutions optimales de sous problèmes. Dans tous les

cas, l'espace mémoire nécessaire à la mémoisation doit être raisonnable (*i.e.,* borné par un polynôme de degré faible).

Dans un algorithme memoisé, la table est initialement vide, ou chacune de ses entrées contient une valeur symbolique dans le cas où le nombre d'entrées maximal est connu. Quand un sous-problème est rencontré la première fois dans l'exécution de l'algorithme récursif, sa solution est calculée et stockée dans la table. À chaque fois que ce sous-problème est rencontré à nouveau, la valeur stockée dans la table est simplement retournée.

L'Algorithme 3 d'évaluation de la suite de Fibonacci est un exemple d'algorithme memoisé. Dans ce cas, on peut utiliser un simple tableau M de taille n initialement rempli de zéros. On considère alors qu'un sous-problème $F(k)$ n'a pas encore été rencontré si $M(k) = 0$, *i.e.,* si la k$^{\text{ième}}$ case du tableau contient un zéro.

Par ailleurs, on remarque que l'espace mémoire nécessaire à la mémoisation n'est pas plus grand que celui occupé par la pile d'exécution de l'algorithme récursif naïf.

Pour reprendre l'exemple de l'exécution d'un Branch and Bound, l'espace mémoire occupé par une structure de donnée si l'on décidait de mémoiser cet algorithme ne pourrait pas être borné. Pour cette raison encore, la programmation dynamique n'est pas applicable pour cet exemple.

4 Exercices.

Exercice 1. Soit l'instance de problème de sac à dos à quatre objets suivante, définie comme dans le cours et le livre support :

$$\begin{array}{ll} v_1 = 3217 & s_1 = 2 \\ v_2 = 4000 & s_2 = 3 \\ v_3 = 2108 & s_3 = 2 \\ v_4 = 1607 & s_4 = 7 \end{array}$$

1. Si $\mu = 100$ dans l'algorithme FPAS-KP, quel est le facteur d'approximation de cet algorithme sur cette instance ?
2. Si l'on veut utiliser l'algorithme FPAS-KP sur cette instance avec une erreur de maximum 1%, sur quelle instance « arrondie » va-t-on lancer l'algorithme PROG-DYN-KP ?

Solution.

1. Rapellons que $\mu = \frac{\varepsilon M}{n}$, or on a $M = 4000$ et $n = 4$, on a donc $\varepsilon = \frac{1}{10}$. Cet algorithme a donc un facteur d'approximation de $1 - \varepsilon = 0.9$. On accepte donc une erreur de l'ordre de 10%.
2. On a $\mu = \frac{1}{100} \cdot \frac{4000}{4} = 10$. On aura donc :

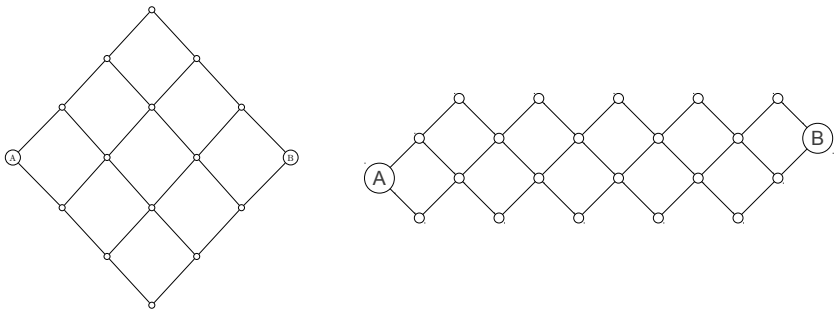
$$\begin{array}{ll} v_1 = 321 & s_1 = 2 \\ v_2 = 400 & s_2 = 3 \\ v_3 = 210 & s_3 = 2 \\ v_4 = 160 & s_4 = 7 \end{array}$$

□

Exercice 2. Soient les deux graphes G et H de la Figure 3. Il est demandé de calculer le nombre de plus courts chemins permettant d'aller de A à B.

1. Concevez un algorithme récursif naïf pour résoudre ce problème.
2. Exhibez la sous-structure d'optimalité de ces problèmes.
3. Illustrez le recouvrement des sous-problèmes.
4. Transformez votre algorithme récursif naïf en algorithme de programmation dynamique en utilisant la mémoisation.

Notez que vos algorithmes doivent fonctionner sur des graphes de taille plus grande que les exemples illustrés. La grille du premier exemple et le serpentin du deuxième peuvent donc avoir une taille n arbitraire.



(a) G (b) H

FIGURE 3 – Deux graphes G et H. Calculez le nombre de plus courts chemins allant de A à B.

Solution.
Soit v un sommet du graphe G ou H, on note
- up_v le prédécesseur supérieur de v,
- $down_v$ le prédécesseur inférieur de v,
- v_{up} le successeur supérieur de v,
- v_{down} le successeur inférieur de v,
- $N(v)$ le nombre de plus courts chemins allant de A à v.

Notez que la notion prédécesseur / successeur est définie en considérant les graphes orientés de gauche à droite (*i.e.,* , on oriente une arête (i, j) de i vers j si i est à gauche de j.). Procéder de cette manière ne change pas le nombre de plus courts chemins de A vers B.[3]

Le calcul de $N(v)$ est donc défini (dans les deux exemples) récursivement comme ci-dessous :

$$N(v) = \begin{cases} 1 & \text{si } v = A, \\ N(down_v) & \text{si } up_v \text{ n'est pas défini,} \\ N(up_v) & \text{si } down_v \text{ n'est pas défini,} \\ N(down_v) + N(up_v) & \text{sinon.} \end{cases}$$

On peut montrer la sous-structure d'optimalité de la façon suivante : soient u et v tels que $u_{up} = v_{down} = w$. Clairement, pour atteindre w à partir de A, il est nécessaire de passer par u ou v. De plus, si $p = A \rightsquigarrow u \longrightarrow w$ est un plus court chemin de A à w, alors $A \rightsquigarrow u$ est un plus court chemin de A à u. En considérant le même argument pour v, le nombre de plus courts chemins de A à w est donc égal à la somme du nombre de plus courts chemins de A à u et de A à v.

On peut particulariser cet argument pour les sommets v aux extrémités de la grille pour lesquels soit v_{up} ou v_{down} n'est pas défini.

Montrons à présent pourquoi l'algorithme récursif est inefficace, et qu'il est amené à calculer plusieurs fois le même résultat, *i.e.,* exhibons le recouvrement des sous-problèmes. En pivotant légèrement le graphe de la Figure 3(a) pour numéroter les sommets (sauf A et B) de gauche à droite et de haut en bas, on obtient l'arbre d'appels récursifs de calcul de $N(B)$ de la Figure 4. On remarque notamment que l'on calcule plusieurs fois les sous-arbres de racine $N(10)$, $N(6)$, etc.

Afin de mémoiser l'algorithme récursif, il est nécessaire d'utiliser une structure de données capable d'enregistrer les valeurs de $N(v)$ pour chaque

3. En réalité, il existe de nombreuses autres paires de sommets u et v pour lesquels considérer ces graphes orientés comme tel ne change pas le nombre de plus courts chemins de u vers v. Voyez-vous lesquels, et pourquoi ?

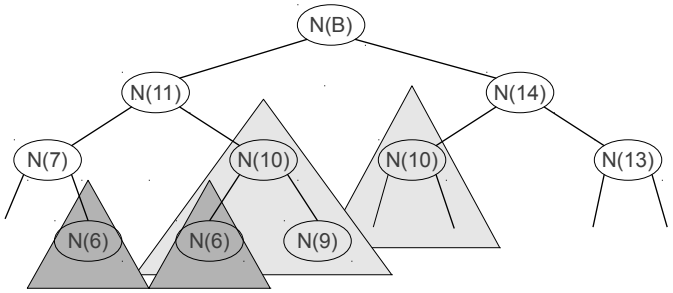


FIGURE 4 – Illustration du recouvrement des sous-problèmes du problème du nombre de plus courts chemins.

sommet v. Étant donné que le nombre de sommets du graphe est fixé avec le problème, on pourrait utiliser un simple tableau indexé judicieusement. Afin de simplifier les notations, on se contentera d'une simple table de hachage T capable d'associer à un sommet v la valeur $N(v)$.

L'algorithme mémoisé de programmation dynamique peut dès lors être conçu de la façon suivante :

Algorithm 3 Algorithme Count-Paths(n)

```
1: T(A) = 1
2: retourner Count(B)

3: Fonction Count(v)
4: si T(v) n'est pas défini
5:     si up_v n'est pas défini
6:         T(v) = Count(down_v)
7:     sinon si down_v n'est pas défini
8:         T(v) = Count(up_v)
9:     sinon
10:        T(v) = Count(up_v) + Count(down_v)
11: retourner T(v)
```

□

Exercice 3. Écrivez un algorithme de programmation dynamique afin de trouver la valeur maximale qui peut être obtenue par un placement approprié de parenthèses dans une expression du type

$$x_1 \ / \ x_2 \ / \ x_3 \ / \ \ldots \ / \ x_{n-1} \ / \ x_n$$

où x_1, ..., x_n sont des nombres rationnels strictement positifs et "/" dénote la division flottante.

Solution.
Soient $1 \leqslant i \leqslant j \leqslant n$, notons $X_{i,j}$ la sous-expression $x_i \ / \ x_{i+1} \ / \ \ldots \ / \ x_n$. De plus, étant donné un parenthésage $P_{1,n}$ de $X_{1,n}$, notons $c(P_{1,n})$ la valeur obtenue en effectuant le quotient en respectant l'ordre donné par les parenthèses. Un parenthésage optimal de $X_{1,n}$ est donc obtenu en maximisant la fonction $c(P_{1,n})$.

Tout parenthésage $P_{1,n}$ de $X_{1,n}$ contient un parenthésage $P_{1,k}$ et $P_{k+1,n}$ des sous-expressions $X_{1,k}$ et $X_{k+1,n}$, pour un certain $1 \leqslant k \leqslant n-1$.[4] On remarque également que cette propriété est également valable récursivement pour $P_{1,k}$ et $P_{k+1,n}$.

La relation de récurrence sur le quotient pour du parenthésage $P_{1,n}$ d'une expression $X_{1,n}$ est donc définie par

$$c(P_{1,n}) = \frac{c(P_{1,k})}{c(P_{k+1,n})}.$$

Par ailleurs, on remarque que tout parenthésage maximal (resp. minimal) $P^*_{1,n}$ de $X_{1,n}$ doit être formé d'un parenthésage $P^*_{1,k}$ qui maximise (resp. minimise) c sur la sous-expression $X_{1,k}$ et d'un parenthésage $P^*_{k+1,n}$ qui minimise (resp. maxmise) c sur la sous-expression $X_{k+1,n}$ pour un certain $1 \leqslant k \leqslant n-1$. En effet, si ce n'était pas le cas, un meilleur parenthésage de $X_{1,k}$ ou $X_{k+1,n}$ conduirait à un meilleur parenthésage $P^*_{1,n}$ de $X_{1,n}$.

Notons $M_{i,j}$ (resp. $m_{i,j}$) la valeur d'un parenthésage maximal (resp. minimal) de $X_{i,j}$ pour $1 \leqslant i \leqslant j \leqslant n$. Sur base des arguments précédents, on peut donc écrire les relations suivantes :

$$\begin{aligned}
M_{i,j} &= m_{i,j} = x_i && \text{si } i = j, \\
M_{i,j} &= \max\left\{ \frac{M_{i,k}}{m_{k+1,j}} \mid i \leqslant k < j \right\} && \text{si } i < j \\
m_{i,j} &= \max\left\{ \frac{m_{i,k}}{M_{k+1,j}} \mid i \leqslant k < j \right\} && \text{si } i < j
\end{aligned}$$

4. Pour simplifier, on ne tient donc ici pas compte de considérations syntaxiques telles que la redondance de parenthèses d'une expression.

On peut dès lors immédiatement en déduire l'algorithme de programmation dynamique suivant :

Algorithm 4 Algorithme MaxChainDivision$(x_1, \ldots, x_n)$

pour $1 \leqslant i \leqslant n$
  $M_{i,i} = m_{i,i} = x_i$
pour $2 \leqslant l \leqslant n$
  pour $1 \leqslant i \leqslant n - l + 1$
    $j = i + l - 1$
    $M_{i,j} = 0$
    $m_{i,j} = \infty$
    pour $i \leqslant k \leqslant j - 1$
      $M_{i,j} = \max\left\{M_{i,j},\ \frac{M_{i,k}}{m_{k+1,j}}\right\}$
      $m_{i,j} = \min\left\{m_{i,j},\ \frac{m_{i,k}}{M_{k+1,j}}\right\}$
retourner $M_{1,n}$

Cet algorithme calcule la valeur d'un parenthésage optimal en $\mathcal{O}(n^3)$.

Notons que s'il on cherche à véritablement parenthéser l'expression, il suffit de maintenir deux tables supplémentaires $s_M(i,j)$ et $s_m(i,j)$ pour $1 \leqslant i, j \leqslant n$, avec $s_M(i,j)$ (resp. $s_m(i,j)$) enregistrant en quel indice k le parenthésage maximal (resp. minimal) de $X_{i,j}$ est séparé en deux parenthésages optimaux de $X_{i,k}$ et $X_{k+1,j}$. On remarque également que maintenir ces tables n'augmente pas la complexité de l'algorithme.

□

Références

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, 1997.

[2] S. Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research*, 50 :48–51, 2002.