Programmation dynamique

Recherche opérationnelle
Dimitri Watel - ENSIIE

2024

La programmation dynamique est une approche de résolution ; une manière de concevoir un algorithme pour résoudre un problème. Ce n'est pas un algorithme.

1 Formule de récurrence

La programmation dynamique est une manière de calculer le résultat d'un algorithme récursif. Pour résoudre un problème avec la méthode de programmation dynamique, il faut d'abord trouver la formule de récurrence que l'on peut calculer avec un algorithme récursif.

On considère dans la suite les trois exemples suivants.

La suite de Fibonacci

Sorties: Le i^e élément de la suite de Fibonacci commençant par 0 et 1.

function $f(i)$
  Si $i \leq 1$ **Alors**
    Renvoyer i
  Renvoyer $f(i-1) + f(i-2)$

La somme de sous-ensemble

Entrées: Deux entiers $i \in [\![0; n]\!]$ et $b \in [\![0; B]\!]$; et n entiers positifs $p_1, p_2, \ldots, p_n$

Sorties: Vérifie s'il existe un sous-ensemble I de $[\![1; i]\!]$ tel que $\sum_{j \in I} p_j = b$

function $g(i, b)$
  Si $i \leq 0$ **Alors**
    Renvoyer $(b = 0)$
  Si $b - p_i \geq 0$ **Alors**
    Renvoyer $g(i-1, b-p_i) \vee g(i-1, b)$
  Sinon
    Renvoyer $g(i-1, b)$

Le plus court chemin dans un DAG

Entrées: Un DAG (graphe sans circuit) G avec des poids ω sur les arcs, deux nœuds v et t de G.

Sorties: Le poids d'un plus court chemin de v à t

function $d(v)$
  Si $v = t$ **Alors**
    Renvoyer 0
  Renvoyer $\min_{u \in \Gamma^+(v)} (d(u) + \omega(v, u))$

2 Mémoïsation

Il existe deux versions de la programmation dynamique, dont la mémoïsation. Cette technique consiste à enregistrer au fur et à mesure les résultats des appels récursifs, de sorte que le calcul n'est pas refait une seconde fois si l'appel est répété.

La mémoïsation est un procédé mécanique, qui peut être exécutée facilement par un ordinateur (par exemple par un compilateur de manière transparente pour l'utilisateur). Nous reprenons ci-après les trois exemples précédents avec la méthode.

Dans chaque algorithme, nous considérons que nous avons accès à une table d'association globale et vide T qui accepte n'importe quel type de clé et de valeur.

La suite de Fibonacci

function $f_{Memo}(i)$
  Si $T(i)$ est vide **Alors**
    Si $i \leq 1$ **Alors**
      $T(i) \leftarrow i$
    Sinon
      $T(i) \leftarrow f_{Memo}(i-1) + f_{Memo}(i-2)$
  Renvoyer $T(i)$

La somme de sous-ensemble

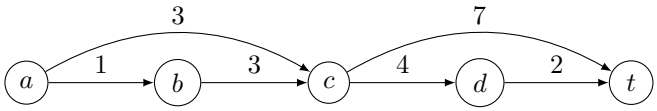
function $g_{Memo}(i, b)$
  Si $T(i, b)$ est vide **Alors**
    Si $i = 0$ **Alors**
      $T(i, b) \leftarrow (b = 0)$
    Sinon Si $b - p_i \geq 0$ **Alors**
      $T(i, b) \leftarrow g(i-1, b-p_i) \vee g(i-1, b)$
    Sinon
      $T(i, b) \leftarrow g(i-1, b)$
  Renvoyer $T(i, b)$

Le plus court chemin dans un DAG

function $d_{Memo}(v)$
  Si $T(v)$ est vide **Alors**
    Si $v = s$ **Alors**
      $T(v) \leftarrow 0$
    Sinon
      $T(v) \leftarrow \min_{u \in \Gamma^+(v)} (d_{Memo}(u) + \omega(v, u))$
  Renvoyer $T(v)$

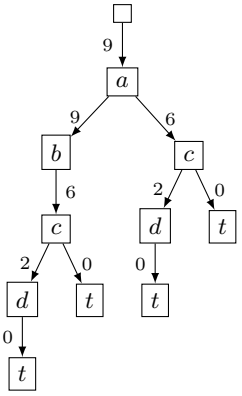
2.1 Exemple et comparaison avec l'algorithme naïf

Montrons un exemple d'exécution sur le plus court chemin. On considère le graphe G suivant dans lequel on cherche le plus court chemin de a à t.



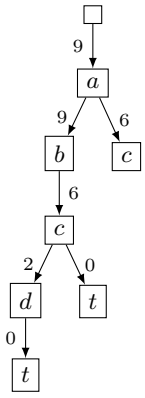
Dans les figures ci-après, chaque arbre représente une exécution et chaque nœud est un appel récursif. On indique sur le nœud la valeur du nœud v donné en entrée. Au dessus d'un nœud, sur son arc entrant, on indique la valeur de retour de l'appel récursif.

Si on applique l'algorithme naïf, on va avoir l'exécution suivante.



On observe un petit défaut de l'algorithme naïf, il appelle plusieurs fois le nœud c et exécute exactement les mêmes appels récursifs deux fois. Dans un graphe quelconque, ceci peut arriver sur de nombreux nœuds et provoquer une explosion combinatoire.

Si on applique la méèhode de la mémoïsation, on aura l'exécution suivante à la place, car au moment où le second appel de c est effectué, on aura déjà stocké le résultat dans la table T. Il n'est donc pas nécessaire de refaire tout le calcul.



2.2 Calcul de la complexité

Dans la suite, on considère comme opération élémentaire les opérations arithmétiques, booléennes, les conditions, le début d'une itération de boucle, les affectations, le fait de renvoyer une valeur et le fait d'appeler une fonction.

Théorème 2.1. *Soit N le nombre d'entrées possibles des appels récursifs et E le nombre maximum d'opérations élémentaires au sein d'un appel récursif. Alors la complexité de l'algorithme de mémoïsation associé est bornée par $O(NE)$.*

Exemple 1. Prenons l'exemple de la somme de sous-ensembles. On a deux entiers en entrée, i et b. L'entier i est entre 0 et n et b est entre 0 et B. Donc $N = (n+1)(B+1) = O(nB)$.

Le nombre d'opérations élémentaires effectuées au maximum par un appel récursif est constant (il n'y a pas de boucle, le nombre d'opérations ne dépend pas de i et b). Donc $E = O(1)$.

Donc la complexité de l'algorithme est donc, d'après ce théorème $O(nB)$.

Proof. Lorsque la fonction est appelée avec une entrée pour la première fois, elle ne peut effectuer plus de E opérations élémentaires au sein de cet appel. Puisqu'il y a N entrées possibles, le nombre total d'opérations élémentaires effectués par les appels de la fonction lorsque l'entrée est vue pour la première fois est $O(NE)$.

A chaque nouvel appel de la fonction, le résultat est enregistré dans T. Donc tout appel effectué avec une entrée déjà vue n'exécutera que les opérations suivantes: vérification de si T contient l'entrée, renvoie de la valeur dans T. Il n'y aura pas de nouvel appel récursif. Ces appels s'effectuent donc en $O(1)$.

Le nombre total d'opérations élémentaires effectués est donc $O(NE)$ plus le nombre d'appels récursifs effectués avec une entrée déjà vue. Il nous reste donc à savoir combien d'appels de la fonction il y a en tout pour calculer la complexité.

Les appels récursifs ne peuvent avoir lieu que lorsque la fonction est appelée avec une entrée pour la première fois. Puisqu'appeler la fonction est une opération élémentaire, alors il ne peut pas y avoir plus de E nouvel appel récursif. Ainsi, la fonction ne peut pas être appelée plus de $O(NE)$ fois.

Donc la complexité est $O(NE + NE) = O(NE)$. □

Ce théorème n'indique qu'une borne supérieure, on peut être un peu plus précis si on dispose d'informations sur l'algorithme.

La mémoïsation est simple à mettre en place mais sa complexité et son temps de calcul réels sont difficiles à évaluer. De plus, elle dépend de l'efficacité du langage à manipuler des fonctions récursives.

3 Méthode itérative

Utiliser la programmation dynamique sous forme iterative pour coder une fonction récursive consiste à résoudre tous les sous-problèmes, en commençant par les cas terminaux et en remontant à l'envers les appels récursifs, jusqu'à ce que le problème initial soit résolu. Les résultats intermédiaires sont stockés dans un tableau.

Il est souvent considéré que cette version est LA méthode de programmation dynamique. Mais notez que la mémoïsation fait la même chose dans le sens inverse. On dit parfois que la mémoïsation est une méthode Top-Down et que la programmation dynamique itérative est une méthode Bottom-up.

Comme dans la mémoïsation, nous supposons que nous avons accès à une table d'association globale et vide T qui accepte n'importe quel type de clé et de valeur. On veut stocker dans T les résultats de tous les appels récursifs nécessaires au calcul. Afin que tout se passe bien, il faut qu'on remonte les appels récursifs: dans T, on stocke le résultat d'un appel seulement si les résultats dont il a besoin pour calculer son propre résultat ont déjà été stockés. Dit autrement, dans l'arbre représentant l'exécution du calcul, il faut remonter les nœuds depuis les feuilles jusqu'à la racine. On va voir cela au travers des trois exemples.

La suite de Fibonacci On cherche à calculer le n-ieme élément de la suite. Chaque appel récursif est $f(i)$ avec i entre 0 et n. On remarque que, lors de l'exécution de $f(i)$, les appels récursifs décrémentent la valeur de i. Si on calcule $f(i-1)$ avant $f(i)$ alors on inverse. Les cas terminaux sont $i = 0$ et $i = 1$. Il suffit donc de calculer $f(i)$ pour i variant de 0 à n.

function $f_{Iter}(n)$
  $T(0) \leftarrow 0$
  $T(1) \leftarrow 1$
  Pour i de 2 à n **Faire**
    $T(i) \leftarrow T(i-1) + T(i-2)$
  Renvoyer $T(n)$

La somme de sous-ensemble On cherche à savoir s'il existe un sous-ensemble I de $[\![1; n]\!]$ tel que $\sum_{j \in I} p_j = B$. Chaque appel récursif est $f(i, b)$ avec i entre 0 et n et b entre 0 et B. On remarque que, lors de l'exécution de $f(i, b)$, les appels récursifs décrémentent la valeur de i. Si on calcule $f(i-1, b)$ (pour tout b) avant $f(i, b)$ (pour tout b aussi) alors on inverse. Les cas terminaux sont $i = 0$, pour tout b. Il suffit donc de calculer, pour i variant de 0 à n, la valeur de $f(i, b)$ pour b variant de 0 à B.

function $f_{Iter}(n, B)$
  Pour b de 0 à B **Faire**
    $T(0, b) \leftarrow (b = 0)$
  Pour i de 1 à n **Faire**
    Pour b de 0 à B **Faire**
      Si $b \geq p_i$ **Alors**
        $T(i, b) \leftarrow T(i-1, b-p_i) \vee T(i-1, b)$
      Sinon
        $T(i, b) \leftarrow T(i-1, b)$
  Renvoyer $T(n, B)$

Le plus court chemin dans un DAG On veut connaître le plus court chemin de s à t dans un DAG de G. Chaque appel récursif est $d(v)$ avec v un nœud de G. On remarque que, lors de l'exécution de $d(v)$, les appels récursifs n'appellent que les successeurs de v. Si on calcule $d(w)$ pour tout successeur w de v avant $d(v)$ alors on inverse. Le cas terminal est $d(t)$. Il suffit donc de calculer $d(v)$ pour tout nœud v en faisant les appels dans un ordre topologique inversé du graphe (un tel ordre est une liste des nœuds où les prédecesseurs d'un nœud w arrivent après w, et un tel ordre existe toujours dans un DAG).

function $d_{Iter}(S)$
  $T(M) = 0$
  $L \leftarrow$ Ordre topologique inverse de $G \backslash M$
  Pour $v \in L$ **Faire**
    $T(v) \leftarrow \min_{u \in \Gamma^+(v)} (T(u) + \omega(v, u))$
  Renvoyer $T(S)$

En conclusion de cette partie, il n'y a pas vraiment de méthode miracle pour mettre en place la méthode itérative. Contrairement à la mémoïsation qui est automatisable, il faut connaître le problème en profondeur. On le constate dans le dernier exemple ou une connaissance de la théorie des graphes est nécessaire pour produire l'algorithme.

Un net avantage de la méthode itérative est qu'il est (généralement) plus facile de déduire la complexité en regardant les algorithmes produits. De plus les algorithmes ne sont pas récursifs, le langage de programmation associé n'a donc pas besoin de gérer efficacement les appels récursifs.

Un défaut comparé à la méthode de mémoïsation est que la méthode itérative produit souvent des algorithmes qui calculent (et stockent) beaucoup plus de résultats que nécessaire, contrairement à la mémoïsation. Par exemple dans la somme de sous-ensemble, l'algorithme itératif rempli T avec $B+1$ valeurs pour chaque entier i. Cependant, en observant l'algorithme récursif, on constate que la mémoïsation ne stockera pas plus d'une valeur quand $i = n$, deux valeurs quand $i = n-1$, quatre quand $i = n-2$, ... Ce défaut peut être parfois partiellement contrecarré en remarquant qu'il n'est pas nécessaire de stocker tout le tableau T, mais cela ne réduit pas la com-

plexité en temps de l'algorithme.

Voici par exemple une variante qui fonctionne très bien pour la somme de sous-ensemble:

function $f_{Iter}(n, B)$
  Pour b de 0 à B **Faire**
    $T(0, b) \leftarrow (b = 0)$
  Pour i de 1 à n **Faire**
    Pour b de 0 à B **Faire**
      Retirer la clef $(i-2, b)$ de T si elle existe.
      Si $b \geq p_i$ **Alors**
        $T(i, b) \leftarrow T(i-1, b-p_i) \vee T(i-1, b)$
      Sinon
        $T(i, b) \leftarrow T(i-1, b)$
  Renvoyer $T(n, B)$

On peut voir qu'en effet $T(i-2, b)$ n'est jamais utilisé, il n'est donc pas nécessaire au calcul.

Remarque 1. On peut voir que les 3 algorithmes naïfs du début du cours sont exponentiels en temps et que les 6 versions de programmation dynamique sont polynomiales (ou presque mais je ne rentre pas dans les détails ici). Ce n'est bien entendu pas une généralité, il existe des problèmes pour lesquels il existe des algorithmes naïfs récursifs exponentiels que la programmation dynamique ne parvient pas à améliorer suffisamment pour les rendre polynomiaux. On peut penser par exemple au problème de voyageur de commerce.