Programmation dynamique

Le principe

La programmation dynamique est un "paradigme" simple et puissant de conception d'algorithmes qu'on peut appliquer pour résoudre un problème si :

1. La solution (optimale) d'un problème de taille n s'exprime en fonction de la solution (optimale) de problèmes de taille inférieure à n -c'est le principe d'optimalité-.
2. Une implémentation récursive "naïve" conduit à calculer de nombreuses fois la solution de mêmes sous-problèmes.

L'idée est alors simplement d'éviter de calculer plusieurs fois la solution du même sous-problème. On définit une table pour mémoriser les calculs déjà effectués : à chaque élément correspondra la solution d'un et d'un seul problème intermédiaire, un élément correspondant au problème final. Il faut donc qu'on puisse déterminer les sous-problèmes qui seront traités au cours du calcul (ou un sur-ensemble de ceux-ci). Ensuite il faut remplir cette table ; il y a deux approches, l'une itérative, l'autre récursive :

- *Version itérative -la classique-*
  On initialise les "cases" correspondant aux cas de base.
  On remplit ensuite la table selon un ordre bien précis à déterminer : on commence par les problèmes de "taille" la plus petite possible, on termine par la solution du problème principal : **il faut bien sûr qu'à chaque calcul, on n'utilise que les solutions déjà calculées**. Le but est bien sûr que chaque élément soit calculé une et une seule fois.
- *Version récursive (tabulation, memoizing)*
  Si l'implémentation classique du principe de programmation dynamique est itérative, une implémentation récursive est possible ! Elle correspond à ce qu'on appelle *memoizing*, i.e. la mémorisation des calculs déjà effectués pour éviter des calculs redondants. A chaque appel, on regarde dans la table si la valeur a déjà été calculée (donc une "case" correspond à un booléen et une valeur ou à une seule valeur si on utilise une valeur "sentinelle"). Si oui, on ne la recalcule pas : on récupère la valeur mémorisée ; si non, on la calcule et à la fin du calcul, on stocke la valeur correspondante. Donc si la fonction f est définie par :

```
fonction f(parametres p)
  si cas_de_base(p) alors g(p) sinon h(f(p_1),...,f(p_k))
```

le schéma de l'algorithme récursif dynamique pour la calculer sera :

```
/* tablecalcul est un dictionnaire contenant les valeurs déjà calculées */
/* on peut  utiliser un tableau, ou une table de hachage */
/* le sous-problème (ou les paramètres le déterminant) est la clé */
fonction fdynrec(parrametres p)
{si non (tablecalcul.contains(p))
   alors /* on calcule et on mémorise */
   {val = (si casdebase(p) alors g(p) sinon h(fdynrec(p_1),...,fdynrec(p_k)));
    tablecalcul. ajouter(p,val);}
 retourner tablecalcul.valeur(p)}
```

Quelques Remarques

L'art du découpage... L'essentiel du travail conceptuel réside dans l'expression d'une solution d'un problème en fonction de celles de problèmes "plus petits". Ensuite, on peut analyser ce qui se passe dans une implémentation récursive "naïve" : si on se rend compte que la solution de mêmes problèmes est recalculée plusieurs fois, on est dans le cadre de la programmation dynamique. Le découpage du problème devrait naturellement conduire à la définition de la table (qui peut être de dimension 1,2,3....) : une case de la table correspond à un sous-problème. Ensuite, il faut encore réfléchir un tout petit peu ... pour savoir comment effectuer le remplissage de la table, tout du moins dans le cas itératif !

Itératif verus Récursif La version récursive est souvent un peu plus simple à écrire. Au niveau de l'efficacité, on peut avoir un surcoût dû à la récursivité, même si l'ordre de grandeur est a priori le même. Par contre, on peut aussi avoir un gain en efficacité dans la version récursive : en effet, on ne calcule que les appels réellement nécessaires, alors que dans le cas itératif, pour certains problèmes, fréquemment on calcule des valeurs inutiles. Donc, pour certaines "configurations" de l'instance du problème, on peut avoir un gain, plus ou moins important.

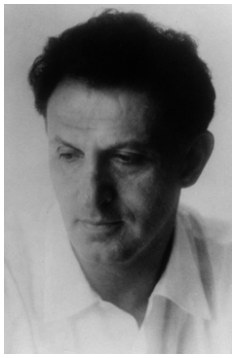
La complexité en temps... Attention, le nombre de sous-problèmes peut être grand : l'algorithme obtenu n'est pas forcément polynômial. Par exemple, dans le cas du "Sac à Dos", il est pseudo-polynômial.

et en espace... En général, la complexité en espace correspond à la table mais il n'est pas forcément nécessaire de mémoriser toutes les valeurs calculées. Dans l'exemple du calcule des C_n^p, seule la ligne du dessus est nécessaire (voire moins). On a par contre souvent besoin de tout stocker quand on doit effectuer une remontée dans la table (voir ci-dessous). Dans les versions récursives, on pourra utiliser des tables de hachage pour essayer d'adapter le mieux possible l'espace mémoire consommé à celui réellement nécessaire.

Le principe d'optimalité

Le principe "la solution optimale d'un problème peut s'obtenir à partir des solutions optimales de sous-problèmes" est appelé dans la littérature *le principe d'optimalité (de Bellman)*. Bien que naturel, le principe n'est pas toujours applicable ! Prenons l'exemple des chemins dans un graphe :

- le principe marche si on cherche le plus court chemin entre deux points (cf algorithme de Bellmann-Ford) : si le chemin le plus court entre A et B passe par C, le tronçon de A à C (resp. de C à B) est le chemin le plus court de A à C (resp. de C à B) ;
- par contre ça ne marche plus si on cherche le plus long chemin sans boucle d'un point à un autre : si le chemin le plus long sans boucle de A à C passe par B, le tronçon de A à B n'est pas forcément le plus long chemin sans boucle de A à B !



Le terme et le concept de programmation dynamique ont été introduits par *Richard Bellmann* pour résoudre des problèmes d'affectation et d'optimisation (recherche opérationnelle) dans les années 50. Le terme “programmation dynamique” peut paraître un peu étrange et est maintenant parfois utilisé dans un autre sens. Programmation, à l'époque, signifiait plutôt planification, le terme "dynamique" venait du fait que les différentes valeurs sont mises à jour dynamiquement. L'histoire dit que Richard Bellman, alors chercheur à la Rand Corporation, devait rendre des comptes au secrétaire à la Défense de l'époque qui n'était un fervent défenseur ni de la “recherche”, ni des "mathématiques" (voir par exemple article de JC Culiol dans la Recherche 2003). Richard Bellmann a alors choisi d'utiliser des termes “programmation” et “dynamique” dans un souci de “marketing”. Un film -*(The) Bellmann equation* - vient de lui être consacré.

Programmation dynamique et Optimisation

La programmation dynamique est souvent utilisée dans le cadre des problèmes d'optimisation : on cherche une solution optimale par rapport à un certain coût (ou fonction objectif) et donc dans ce cas c'est non seulement le meilleur coût qu'on cherche, mais bien sûr aussi une solution qui corresponde à ce coût. Or souvent, la table construite dans le cadre de la programmation dynamique contient des coûts et non des solutions. Dans une première phase, on calcule donc seulement le coût d'une solution optimale ; il est cependant souvent facile ensuite de récupérer la (une) solution optimale, toujours en utilisant l'information contenue dans la table (éventuellement en enrichissant un petit peu la table : on peut mémoriser pour une valeur d'où elle vient...) ; grosso modo, on parcourt la table à l'envers par rapport à sa construction, i.e. en partant de la case correspondant au problème principal et en "remontant" vers les plus petits problèmes : la solution correspond au chemin dans la table. On appelle souvent ce processus "la remontée dans la table" et la matrice enrichie une matrice de “remontée”.

Quelques exemples classiques

Beaucoup d'algorithmes dans des domaines divers - recherche opérationnelle, apprentissage, bio-informatique, contrôle de systèmes, linguistique, théorie des jeux, processus de Markov, stratégie d'investissement, ... utilisent le principe de la programmation dynamique. Citons quelques exemples : l'algorithme de Warshall-Floyd (clôture transitive d'une relation), l'algorithme de Bellmann-Ford (plus court chemin dans un graphe), l'algorithme de Viterbi, le calcul de la distance de deux chaînes (cf *diff* d'Unix) et plus généralement les problèmes d'alignement de séquences, l'algorithme de Cocke-Younger-Kasami (CYK), l'algorithme d'Earley (analyse syntaxique de phrases), les arbres binaires optimaux, la triangulation d'un polygone....

Evaluation paresseuse et programmation dynamique

Dans des langages fonctionnels non stricts (comme Haskell), on peut utiliser l'évaluation paresseuse : une valeur n'est calculée que si vraiment on en a besoin ("call by need"). Cela permet de programmer de façon très simple des algorithmes de type "programmation dynamique". On définit la table avec les valeurs d'initialisation et la formule de récurrence ; si on demande le "calcul" d'une valeur, ne seront calculées -et qu'une seule fois- que les valeurs nécessaires. Par exemple soit le programme Haskell :

```
coeefbin  n k=  --precondition 0 =< k <= n
   let
   cb= array ((0,0, (n,k)) (
   [((i,0),1) | i<- [0..n]]  --cb(i,0)=1, 0<=i<=n
   ++
   [((i,i),1) | i<- [1..k]]  --cb(i,i)=1, 1<=i<=k
   ++
   [((i,j),cb!(i-1,j-1) + cb!(i-1, j)) | i<- [2..n],j<-[1..k], (j<i)]
   --cb(i,j)=cb(i-1,j-1) + cb(i-1, j) i>j>0  )
 in
   cb!(n,k)
```

Il ne calculera qu'une et une seule fois les valeurs de $cb(i,j)$ nécessaires pour le calcul de $cb(n,k)$!

Programmation dynamique et graphes : Graphe des sous-problèmes, tri topologique, plus court chemin...

Le graphe des sous-problèmes

On peut donc associer à un problème un graphe dirigé : un sommet du graphe correspond à un sous-problème. Un arc entre deux sommets A et B exprime que le sous-problème A dépend directement du sous-problème B. Bien sûr, a priori, le graphe doit être acyclique ... Un ordonnancement du calcul des sous-problèmes correspond alors à trouver un tri topologique des sous-problèmes : on ne calcule un sous-problème que si les sous-problèmes dont il dépend ont été résolus.

Le plus court chemin

Revenons sur le problème du plus court chemin d'une source s à un sommet d'un graphe v, pour tout v. Plaçons-nous dans le cas d'un graphe pondéré et notons $c(w,v)$ le poids de l'arc de v à w (égal à $+\infty$ si il n'y a pas d'arc). Soit $Dist(v)$ la distance de s à v. On a alors :

$$Dist(s) = 0, Dist(v) = min_{(w,v)arc}(Dist(w) + c(w,v))$$

Si le graphe est acyclique, par *memoization*, on obtient directement un algorithme récursif. Toujours dans le cas acyclique, un algorithme itératif de programmation dynamique correspond dans ce cas à calculer les sous-problèmes $Dist(w)$ en énumérant les w selon un tri topologique à partir de s.

```
Dist[s]=0;
pour tout sommet v dans un ordre de tri topologique
    pour tout arc (w,v)
       Dist[v]=min (Dist[v],Dist[w]+c[w][v]).
```

Si le graphe est cyclique mais si tous les poids sont positifs, on peut remarquer qu'un chemin minimal passe par au plus n sommets, si n est le nombre de sommets. En notant $Dist(i, v)$ la longueur minimale d'un chemin de s à v qui passe par au plus i sommets, on obtient :

$Dist(1, v) = 0$ si $s = v$, $= +\infty$ sinon.

$Dist(i, v) = min(Dist(i-1, v), min_{(w,v)arc}(Dist(i-1, w) + c(w, v))$.

On a donc l'algorithme suivant en utilisant une table pour stocker les $Dist(i, j)$:

```
pour tout sommet v Dist[1][v]=+∞;
Dist[1][s]=0;
pour i de 1 à n
    pour tout arc (w,v) Dist[i][v]=min(Dist[i-1][v],Dist[i-1][w]+c[w][v]).
```

On remarque ici qu'on est dans le cas où il n'est pas nécessaire de mémoriser tous les $Dist(j, v)$ puisqu'on utilise juste les précédents. On obtient donc :

```
pour tout sommet v  Dist[v]=+∞;
Dist[s]=0;
Pour i de 1 à n
    pour tout arc (w,v) Dist[v]=min(Dist[v], Dist[w]+c[w][v]).
```

Si on veut de plus mémoriser le chemin, on peut utiliser une autre table Pr :

```
pour tout sommet v Dist[v]=+∞;
Dist[s]=0;
Pour i de 1 à n
    pour tout arc (w,v)
             si (Dist[v]>Dist[w]+c[w][v]) {Dist[v]=Dist[w]+c[w][v]);Pr[v]=w;}
```

On retrouve ainsi l'algorithme de Bellmann-Ford.

Réciproquement, trouver la solution optimale d'un problème peut souvent se ramener à chercher un chemin optimal dans le graphe des sous-problèmes.

D'un paradigme à l'autre : Programmation dynamique, Algorithmes gloutons et "Diviser pour Régner"

"Diviser pour régner" repose sur la décomposition du problème en problèmes de taille substantiellement plus petites, typiquement en problèmes de taille divisée par 2, 3, ..., sous-problèmes souvent "disjoints". Dans le cadre de la programmation dynamique, on réduit souvent le problème en sous-problèmes de taille "légèrement plus petite". Par contre, ces sous-problèmes sont souvent non disjoints, i.e. ils partagent des sous-problèmes.

Si on représente l'arbre de décomposition d'un problème, dans le cas "Divide and Conquer", l'arbre sera souvent de hauteur logarithmique par rapport à la taille de la donnée, d'où l'efficacité a priori de la démarche. Dans le cas de la programmation dynamique, l'arbre est souvent de profondeur linéaire par rapport à la taille du problème et a donc potentiellement un nombre exponentiel de noeuds mais de nombreux noeuds correspondent au même sous-problème et l'arbre peut donc être représenté de façon beaucoup plus compacte comme un graphe (DAG : directed acyclic graph).

Toujours avec cette représentation de l'arbre de décomposition du problème, typiquement un algorithme glouton n'explore qu'une seule branche de l'arbre.