Programmation Dynamique

ACT

Sophie Tison
Master Informatique
Université de Lille

Trois paradigmes d'algorithmes

Algorithmic Design Pattern

- Rappels
- Paradigmes d'algorithme
  - Diviser pour régner
  - Programmation Dynamique
  - Algorithmes Gloutons
- Complexité des problèmes
- Algorithmique Avancée

Quand peut-on utiliser la programmation dynamique?

- 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-.
- Une implémentation récursive "naïve" conduit à calculer de nombreuses fois la solution de mêmes sous-problèmes.

Le principe d'optimalité

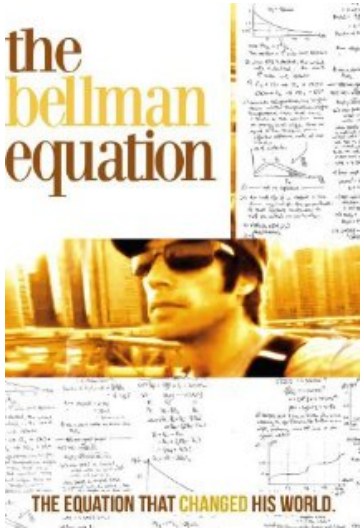
- si on cherche le plus court chemin entre deux points (cf algorithme de Floyd): si le chemin le plus court entre A et B passe par C, le tronçon de A à C (resp. de C à B) est le chemin le plus court de A à C (resp. de C à B); le principe est respecté.
- si on cherche le plus long chemin sans boucle d'un 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 principe n'est pas respecté.

Richard Bellman

Le concept de programmation dynamique a été introduit par Richard Bellman pour résoudre des problèmes de recherche opérationnelle dans les années 50.
Le terme “programmation dynamique” - les différentes valeurs sont mises à jour dynamiquement- a été choisi dans un souci de “marketing”.

The Bellman Equation

retrace la vie de Richard Bellman.



Richard Bellman

extrait du site du film

”One of the most influential mathematicians of the 20th century, Mr. Bellman was a pioneer in the field of computer science. Bellman’s work changed the perception of the application of mathematics within science. The term ’Bellman Equation’ is a type of problem named after its discoverer, in which a problem that would otherwise be not possible to solve is broken into a solution based on the intuitive nature of the solver. Applied in control theory, economics, and medicine, it has become an important tool in using math to solve really difficult problems. A fiercely independent and controversial figure, Mr. Bellman published more than 40 books and 600 papers before his early death of a brain tumor. The inventor of the field of dynamic programming was persecuted by McCarthy, worked on the Manhattan project, and remained as much a mystery as the problems he conquered.”

Comment utiliser la programmation dynamique?

On définit une table pour mémoriser les calculs déjà effectués: à chaque élément correspondra la solution -souvent plutôt le coût ou le bénéfice- d’un et d’un seul problème intermédiaire, un élément correspondant au problème final.
Il faut donc qu’on puisse déterminer les sous-problèmes (ou un sur-ensemble de ceux-ci) 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.
Il y a souvent une dernière étape, ”la remontée dans la table”, pour récupérer la solution et non pas seulement le coût ou le bénéfice de celle-ci.

La version itérative

- 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.

La version récursive (tabulation, memoizing)

- A chaque appel, on regarde si la valeur a déjà été calculée (soit en utilisant une table de booléens ou une valeur ”sentinelle” dans la table des valeurs).
- Si oui, on récupère la valeur mémorisée.
- Si non, on la calcule, on mémorise qu'on l'a calculée et la valeur correspondante.

La version récursive (tabulation, memoizing)

Donc si la version récursive naïve est

```
fonction f(parametres p)
if cas_de_base(p) then
 |  return (g(p))
else
 |  return h(f(p_1), ..., f(p_k))
```

le schéma de l'algorithme récursif dynamique sera

```
//tabcalcul: dictionnaire des valeurs calculées
// le sous-problème-ses paramètres- est la clé
fonction fdynrec(parametres p)
if cas_de_base(p) then
 |  return g(p);
if non (tabcalcul.contains(p)) then
 |  val = h(fdynrec(p_1), ..., fdynrec(p_k)));
 |  tabcalcul.ajouter(p,val);
return tabcalcul.valeur(p) ;
```

Conception d'un algorithme de programmation dynamique

L'essentiel du travail conceptuel réside dans l'expression d'une solution d'un problème en fonction de celles de problèmes ”plus petits”!

Principe utilisé dans beaucoup d'algorithmes de domaines divers

Recherche opérationnelle, apprentissage, bio-informatique, commande de systèmes, linguistique, théorie des jeux, processus de Markov....

- algorithme de Warshall-Floyd
- calcul de la distance de deux chaînes (cf diff d'Unix), problèmes d'alignement de séquences
- algorithme de Cocke-Younger-Kasami, algorithme d'Earley (analyse syntaxique de phrases)
- calcul d'arbres binaires optimaux
- triangulation d'un polygone, …
- algorithme de Viterbi, modèles cachés de Markov
- apprentissage par renforcement, …

Un autre exemple: la plus longue sous-suite commune

Le problème: Une sous-suite (ou sous-mot) d'un mot u est un mot w obtenu à partir de u en effçant des lettres:
act est sous-suite de pacte, de tract, mais aussi de accepter, rachat, lancement, ...
Soient deux mots u et v. On cherche la longueur de la (ou d'une) plus longue sous-suite commune des deux mots ainsi qu'une telle sous-suite. On notera $lcs(u, v)$ cette longueur.

Exemple: si u = plage et v = algo, ag et lg sont les deux sous-suites de longueur maximale et donc $lcs(u, v)$ vaut 2.

Analyse du problème

Les sous-problèmes

Soit $n = |u|$, $p = |v|$, $u = u_1...u_n, v = v_1..v_p$
Notons $LCS(i, j)$ la longueur maximale d'une sous-suite des mots $u_1...u_i$ $(0 \leq i \leq n)$ –les i premières lettres de u, et $v_j..v_{p-1}$ $(0 \leq j \leq p)$ –les j premières lettres de v

Analyse du problème

La résolution

On a donc:

- Les cas de base: $LCS(i, 0) = 0 = LCS(0, j)$ –un des deux mots est vide
- la récurrence:
  - si $u_i = v_j$, $LCS(i, j) = 1 + LCS(i - 1, j - 1)$
  - si $u_i \neq v_j$, $LCS(i, j) = \max(LCS(i - 1, j), LCS(i, j - 1))$

Le cœur de l'algorithme

- $LCS(i,0) = 0 = LCS(0,j), 0 \leq i \leq |u|, 0 \leq j \leq |v|$
- $1 \leq i \leq |u|, 1 \leq j \leq |v|$
  si $u_i = v_j$ $LCS(i,j) = 1 + LCS(i-1, j-1)$
  sinon $LCS(i,j) = \max(LCS(i-1,j), LCS(i,j-1))$

Version récursive naïve

Input: u,v deux mots
Output: retourne LCS(i,j) pour u et v

```
int solvpart(int i, int j)
if ((i == 0)||(j == 0)) then
 |  return 0;
else if (u.charAt(i − 1) == v.charAt(j − 1)) then
 |  return 1+solvpart(i-1, j-1);
else
 |  return max(solvpart(i-1,j),solvpart(i, j-1));
```

Complexité de la version récursive naïve?

Dans le pire des cas, le nombre d'appels est au moins de l'ordre de $2^{\min(u.length(),v.length())}$.

Dans le meilleur des cas, le nombre d'appels est $\min(u.length(), v.length())$.

Complexité de la version récursive naïve?

La complexité dans le pire des cas (en nombre d'appels) est au moins de l'ordre de $2^{\min(u.length(),v.length())}$

Le nombre d'appels différents est au plus $(1 + |u|) * (1 + |v|)$.

On est dans le cadre de la programmation dynamique!

On va utiliser une table à deux dimensions indexée par (i, j) avec $0 \leq i \leq |u|, 0 \leq j \leq |v|$.

Version dynamique itérative

Initialisation

- $LCS(i,0) = 0 = LCS(0,j), 0 \leq i \leq |u|, 0 \leq j \leq |v|$
- $1 \leq i \leq |u|, 1 \leq j \leq |v|$
  si $u_i = v_j$ $LCS(i,j) = 1 + LCS(i-1, j-1)$
  sinon $LCS(i,j) = \max(LCS(i-1,j), LCS(i,j-1))$

```
int T[][]=new int[u.length()+1][v.length()+1];
//T[i][j] memorisera LCS(u[0..i-1],v[0,..j-1])
for (int i = 0; i ≤ u.length(); i + +) do
 |  T[i][0] = 0;
end
for (int j = 0; j ≤ v.length(); j + +) do
 |  T[0][j] = 0;
end
...
```

Version dynamique itérative

La récurrence

- $LCS(i,0) = 0 = LCS(0,j), 0 \leq i \leq |u|, 0 \leq j \leq |v|$
- $1 \leq i \leq |u|, 1 \leq j \leq |v|$
  si $u_i = v_j$ $LCS(i,j) = 1 + LCS(i-1, j-1)$
  sinon $LCS(i,j) = \max(LCS(i-1,j), LCS(i,j-1))$

```
...
for (int i = 1; i ≤ u.length(); i++) do
    for (int j = 1; j ≤ v.length(); j++) do
        if (u.charAt(i − 1) == v.charAt(j − 1)) then
            T[i][j] = 1 + T[i − 1][j − 1];
        else
            T[i][j] = max(T[i][j − 1], T[i − 1][j]);
    end
end
...
```

Version dynamique itérative

```
int T[][]=new int[u.length()+1][v.length()+1];
//T[i][j] memorisera LCS(u[0..i-1],v[0,..j-1])
for (int i = 0; i ≤ u.length(); i++) do
    T[i][0] = 0;
end
for (int j = 0; j ≤ v.length(); j++) do
    T[0][j] = 0;
end
for (int i = 1; i ≤ u.length(); i++) do
    for (int j = 1; j ≤ v.length(); j++) do
        if (u.charAt(i − 1) == v.charAt(j − 1)) then
            T[i][j] = 1 + T[i − 1][j − 1];
        else
            T[i][j] = max(T[i][j − 1], T[i − 1][j]);
    end
end
return T[pb.u.length()][pb.v.length()] ;
```

Version dynamique itérative

```
...
for (int i = 0; i ≤ u.length(); i++) do
    T[i][0] = 0;
end
for (int j = 0; j ≤ v.length(); j++) do
    T[0][j] = 0;
end
for (int i = 1; i ≤ u.length(); i++) do
    for (int j = 1; j ≤ v.length(); j++) do
        if (u.charAt(i − 1) == v.charAt(j − 1)) then
            T[i][j] = 1 + T[i − 1][j − 1];
        else
            T[i][j] = max(T[i][j − 1], T[i − 1][j]);
    end
end
return T[pb.u.length()][pb.v.length()] ;
```

Complexité en $O(|u| * |v|)$.

La remontée, ou comment récupérer une sous-suite commune de longueur maxi

```
Input: TCalc[i][j]=LCS(i,j) pour les valeurs "nécessaires"
String s =""; //s contiendra une sous-suite maxi
int i = u.length(), j = v.length();
//(i,j) représente la "case courante"
// on part de la case "finale" et remonte jusqu'au cas de base
while ((i > 0)&&(j > 0)) do
    if (u.charAt(i − 1) == v.charAt(j − 1)) then
        s = u.charAt(i − 1) + s;
        i − −; j − −;
    else if (T[i][j] == T[i − 1][j]) then
        i − −;
    else
        j − −;
end
return s;
```

La remontée, ou comment récupérer une sous-suite commune de longueur maxi

```
if (u.charAt(i − 1) == v.charAt(j − 1)) then
...
else if (T[i][j] == T[i − 1][j]) then
 |  i − −;
else
 |  j − −;
```

ou

```
if (u.charAt(i − 1) == v.charAt(j − 1)) then
...
else if (T[i][j] == T[i][j − 1]) then
 |  j − −;
else
 |  i − −;
```

Version récursive dynamique?

Version récursive "naïve" de solvpart(int i, int j)

```
if ((i == 0)||(j == 0)) then
 |  return (0);
else if (u.charAt(i − 1) == v.charAt(j − 1)) then
 |  return (1+solvpart(i-1, j-1));
else
 |  return ((solvpart(i-1,j),solvpart(i, j-1)));
```

Version récursive dynamique , solvpartDyn(int i, int j)

```
T[][]=new int[u.length()+1][v.length()+1]  initialisée à -1
if ((i == 0)||(j == 0)) then
 |  return 0;
if (T[i][j] == −1) then
 |  if (u.charAt(i − 1) == v.charAt(j − 1)) then
 |   |  T[i][j]=1+solvpartDyn(i-1, j-1);
 |  else
 |   |  T[i][j]=max(solvpartDyn(i-1,j),solvpartDyn(i, j-1));
return T[i][j];
```

Complexité de la version récursive dynamique

```
...
int solvpartDyn(int i, int j)
if ((i == 0)||(j == 0)) then
 |  return 0;
if (T[i][j] == −1) then
 |  if (u.charAt(i − 1) == v.charAt(j − 1)) then
 |   |  T[i][j]=1+solvpartDyn(i-1, j-1);
 |  else
 |   |  T[i][j]=max(solvpartDyn(i-1,j),solvpartDyn(i, j-1));
return T[i][j];
```

Un T[i][j] est calculé au plus une fois et génère au plus deux appels à solvpart.

Le nombre d'appels est en $O(|u| \times |v|)$.

Remarque: si les mots sont identiques, le nombre d'appels est en $O(|u|)$.

Un dernier exemple: le partage

Le but est de diviser l'intervalle $[0, m]$ en k morceaux en minimisant le coût du découpage.

Le coût du découpage est ici la somme des carrés des longueurs des morceaux: si les longueurs des morceaux sont $y_1, ...y_k$, on veut minimiser $\sum_{i=1}^{k} y_i^2$.

On impose la contrainte que les découpes soient effectuées à des endroits "prédécoupés" $0 < x_1 \ldots < x_n < m$.

le coût?

Le but est de diviser l'intervalle $[0, m]$ en k morceaux en minimisant le coût du découpage. Le coût du découpage est ici la somme des carrés des longueurs des morceaux: si les longueurs des morceaux sont $y_1, ...y_k$, on veut minimiser $\sum_{i=1}^{k} y_i^2$.
Si il n'y a aucune contrainte, il suffit de découper en k parties égales. Pourquoi?

En minimisant la somme des carrés, on exprime qu'on veut "équilibrer" les morceaux.

Formalisation du problème

Entrée:

m, entier, la taille de l'intervalle

k, entier, le nombre de morceaux souhaités

$n >= k - 1$, entier, le nombre de prédécoupes

$0 < x_1 \ldots < x_n < m$ les prédécoupes.

Sortie:

$k - 1$ points parmi les n points x_i, $x_{i_1} < x_{i_2} < \ldots < x_{i_{k-1}}$,

qui minimisent le coût du découpage associé -la fonction objectif-soit $\sum_{j=1}^{k}(x_{i_j} - x_{i_{j-1}})^2$.
(On pose $x_{i_0} = 0, x_{i_k} = m$)

Parenthèse: les problèmes d'optimisation

On est dans un cadre classique: celui des problèmes d'optimisation.

On cherche une solution correcte - ici un découpage qui respecte les contraintes- qui minimise un coût (maximise un bénéfice).

Ici, combien de solutions possibles? $\binom{n}{k-1}$ $(=C_{k-1}^{n})$

Ce nombre peut être très grand; par exemple, $\binom{2p}{p} \geq 2^p$

Décomposition du problème: Les cas simples?

$k = 1$ Le coût est alors m^2.

$n = k - 1$ On n'a pas le choix.

Comment construire une solution

On choisit un x_i et on découpe à gauche du x_i en $k-1$ parties.
Remarque fondamentale: Soit un découpage optimal en k parties dont le dernier élément est x_i donne un découpage optimal en $k-1$ parties de $[0, x_i]$;

C'est le principe d'optimalité.
La solution optimale d'un problème s'exprime en fonction de solution optimale de sous problèmes.

Comment construire une solution

Donc un sous-problème est défini par deux paramètres i et j.
$P(i,j)$ est le découpage découpage optimal de $[0, x_i]$ en j éléments, avec $1 \leq j \leq i \leq n+1$.
On a donc:
$P(i,1) = x_i^2$
$P(i,j) = Min_{j-1<=l<i}(P(l,j-1) + (x_i - x_l)^2)$ si $2 \leq j \leq i$

D'où l'algo "naïf"

$P(i,1) = x_i^2$
$P(i,j) = Min_{j-1<=l<i}(P(l,j-1) + (x_i - x_l)^2), 2 \leq j \leq i$

```
int P(int i, int j) {
if (j==1) return x[i]*x[i];
else {
    int res=MaxInt;
    for (int l=j-1; l <i;l++)
       res=min(res, P(l,j-1)+(x[i]-x[l])*(x[i]-x[l]));}
     return res;   }
```

Complexité?

D'où l'algo "naïf"

$P(i,1) = x_i^2$
$P(i,j) = Min_{j-1<=l<i}(P(l,j-1) + (x_i - x_l)^2), 2 \leq j \leq i$

```
int P(int i, int j) {
if (j==1) return x[i]*x[i];
else {
    int res=MaxInt;
    for (int l=j-1; l <i;l++)
       res=min(res, P(l,j-1)+(x[i]-x[l])*(x[i]-x[l]));}
     return res;   }
```

Une branche de l'arbre des appels pour P(n+1,k)=un découpage possible, donc nombre d'appels au moins $\binom{n}{k-1}$.

En programmation dynamique

$P(i,1) = x_i^2$
$P(i,j) = Min_{j-1<=l<i}(P(l,j-1) + (x_i - x_l)^2), 2 \leq j \leq i$

```
int P[][]=new int[n+2][k+1];
......
for (int i=1; i<=n+1;j++)
  P[i][1]=x[i]*x[i];
  for (int j=2; j<=k && j <=i;j++) {
    P[i][j]=MaxInt;
    for (int l=j-1; l <i;l++)
      P[i][j]= min(P[i][j] , P[l][j-1]+(x[i]-x[l])*(x[i]-x[l]));}

 return P[n+1][k];
```

Programmation dynamique: bilan

Quand?

- 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-.
- Une implémentation récursive "naïve" conduit à calculer de nombreuses fois la solution de mêmes sous-problèmes.

Comment? En mémorisant les solutions - seulement leur fonction objectif en général- des sous-problèmes.

L'essentiel du travail conceptuel réside dans l'expression d'une solution d'un problème en fonction de celles de problèmes "plus petits".

Où découper?

```
int P[][]=new int[n+2][k+1];
......
for (int i=1; i<=n+1;j++)
  P[i][1]=x[i]*x[i];
  for (int j=2; j<=k && j <=i;j++) {
    P[i][j]=MaxInt;
    for (int l=j-1; l <i;l++) {
      if (P[i][j] > P[l][j-1]+(x[i]-x[l])*(x[i]-x[l]))
      {P[i][j] = P[l][j-1]+(x[i]-x[l])*(x[i]-x[l]));
        Dec[i][j]=l;}
 //remontée
 i=n+1;
 for (int j=k; j>1;j--){
   découper en Dec[i][j];
    i=Dec[i][j];
    }
```