Programmation Dynamique

AAC

Sophie Tison-Lille 1-Master1 Informatique

Trois paradigmes

- Diviser Pour Régner
- Algorithmes gloutons
- Programmation Dynamique

La programmation dynamique est :

Un schéma d’algorithme exhibé dans les années 1950 par Bellman et basé sur deux idées simples.

.Résoudre un problème grâce à la solution de sous-problèmes

.Eviter de calculer deux fois la même chose, i.e. la solution du même sous-problème.

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- appelé parfois principe de Bellmann.

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

Comment utiliser la programmation dynamique?

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.

Ensuite il faut remplir cette table soit itérativement, soit récursivement.

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”!!!

Un exemple de conception: 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 (c.a.d. 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" $0 < x_1 \ldots < x_n < m$.

Remarque: Si il n'y a aucune contrainte, il suffit de découper en k parties égales. (Preuve??)

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 prédécoupages.

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 $x_{i_0} = 0, x_{i_k} = m$)

On est dans le cadre classique des problèmes d'optimisation.

Parenthèse: la construction d’une solution et de l’algorithme

- Souvent les solutions sont des suites de choix.
- On peut représenter l’espace des solutions comme un arbre, l’arbre des choix où un noeud représente une solution partielle, un sous-problème.
- Cadre de la programmation dynamique: plusieurs noeuds de l’arbre correspondent aux mêmes sous-problèmes
- Exemple d’un jeu: un noeud = une configuration: plusieurs suites de coup peuvent aboutir sur la même configuration.
- L’algorithme ”s’appuie sur cet arbre”;

Retour à l'exemple: 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 (le dernier) x_i et on découpe à gauche du x_i en $k-1$ parties. ou on choisit un (le premier) x_i et on découpe à droite du x_i en $k-1$ parties.
Remarque fondamentale: Soit un découpage optimal en k parties dont le dernier élément est x_i: il 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 peut être 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:

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

```
int P[][]=new int[n+2][k+1];
......
for (int i=1; i<=n+1;j++)
  P[i][1]=x[i]^2;
  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])^2);}

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

Comment savoir où découper?

```
int P[][]=new int[n+2][k+1];
......
for (int i=1; i<=n+1;j++)
  P[i][1]=x[i]ˆ2;
  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])ˆ2)
      {P[i][j] = P[l][j-1]+(x[i]-x[l])ˆ2);
        Dec[i][j]=l;}
 //remontée
 i=n+1;
 for (int j=k; j>1;j--){
   découper en Dec[i][j];
    i=Dec[i][j];
    }
```

Programmation dynamique et plus court chemin dans un graphe

On peut souvent voir un problème comme un problème de plus court chemin dans un graphe qui correspond à la table des sous-problèmes. Ici:

- une case (=un sous-problème) correspond à un noeud
- Il y a un arc de la case (i, j) vers la case $(l, j-1)$, $j-1 \leq l < i$ de poids $(x[i] - x[l])^2$
- Il y a un arc de la case $(i, 1)$ vers la case FIN, de coût x_i^2.
- On cherche un chemin de poids minimal depuis $(n+1, k)$ jusqu'à la case fin.

Les algos de plus court chemin dans un graphe?

- Un algorithme glouton: Dijkstra, dans le cas des poids positifs
- Un algorithme de type programmation dynamique: Bellmann-Ford dans le cas des poids quelconques pour les graphes acycliques ou sans cyle négatif.

Plus court chemin

Si Dist(s,n) est le plus court chemin de s à t

- $Dist(s,s) = 0$
- $Dist(s,t) = min_{(n,t)arc}(Dist(s,n) + val(n,t))$ avec $val(n,t)$: valeur de l'arc de n à t

Attention aux cycles!!!

$$Dist(s,s) = 0$$
$$Dist(s,t) = min_{(n,t)arc}(Dist(s,n) + val(n,t))$$

Si le graphe est sans cycle, on en déduit un algo récursif:

```
Dist(s,t)
si s==t
alors return 0
sinon return min_{arc(n,t)}(Dist(s,n) + val(n,t))
```

Mais pour éviter de faire plusieurs fois le même calcul, on mémorise chaque valeur Dist(u,v) !!!

Attention aux cycles!!!

$Dist(s, s) = 0$
$Dist(s, t) = min_{(n,t)arc}(Dist(s, n) + val(n, t))$

Pour faire une version itérative, dans quel ordre énumérer les noeuds?

Un noeud ne peut être traité que si tous ses prédécesseurs ont été traités:
Tri topologique!!!

Schéma d'Algo:

```
Initialiser Dist(s,s) à 0;
Pour t dans l'ordre d'un tri topologique
```

$\quad Dist(s, t) = min_{arc(n,t)}(Dist(s, n) + val(n, t)))$

le Principe d'optimalité

- Si le chemin le plus court de a à b passe par c, de a à c il emprunte le chemin le plus court.
- Soit le problème du plus long chemin sans cycle dans un graphe avec des cycles:
  Si le chemin le plus long sans cycle de a à b passe par c, de a à c il n'emprunte pas forcément le chemin le plus long sans cycle!!!
- Dans un cas, le principe d'optimalité est respecté, pas dans l'autre!!!

Le cas cyclique

Si le graphe est cyclique mais si tous les poids sont positifs ou si il n'y a pas de cycle négatif, 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.

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

Le cas cyclique

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 2 à n
 pour tout arc (w,v)
  Dist[i][v]=min(Dist[i-1][v],Dist[i-1][w]+c[w][v])
```

Complexité temporelle?

```
pour tout sommet v Dist[1][v]=+∞;
Dist[1][s]=0;
pour i de 2 à n
 pour tout arc (w,v)
  Dist[i][v]=min(Dist[i-1][v],Dist[i-1][w]+c[w][v])
```

Complexité temporelle: $O(n * m)$ si m est le nombre d'arcs.

Complexité spatiale?

```
pour tout sommet v Dist[1][v]=+∞;
Dist[1][s]=0;
pour i de 2 à n
 pour tout arc (w,v)
  Dist[i][v]=min(Dist[i-1][v],Dist[i-1][w]+c[w][v])
```

Complexité spatiale: ici $O(n^2)$. Peut-on faire mieux?

Economiser l'espace

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[1][v]=+∞;
Dist[1][s]=0;
pour i de 2 à n
 pour tout arc (w,v)
  Dist[v]=min(Dist[v],Dist[w]+c[w][v])
```

Complexité spatiale: ici $O(n)$.

Améliorer la complexité temporelle?

Les $n-1$ boucles ne sont pas toujours nécessaires!

```
pour tout sommet v Dist[1][v]=+∞;
Dist[s]=0;
Modifie=true;
Tant que Modifie{
 pour tout arc (w,v)
  si (Dist[v]>Dist[w]+c[w][v])
   {Dist[v]=Dist[w]+c[w][v];Modifie=true;}}
```

Si il n'y a pas de cycle négatif, s'arrêtera après au plus $n-1$ boucles. Complexité dans le pire des cas est toujours $O(n^2)$, dans le meilleur des cas en $0(1)$.

Le cas cyclique

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 2 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.

Programmation dynamique, Décision , Equation de Bellmann

La programmation dynamique est souvent utilisée pour déterminer une stratégie optimale.
Comme on l'a déjà mentionné, on peut souvent modéliser le système par un graphe:

- un sommet est un état ou une configuration du système
- à chaque sommet/état s et associé un ensemble d'actions $Act(s)$ (ou de choix, de décisions).
- à un état s et une action a sont associés un nouvel état $T(s,a)$ et une valeur $v(s,a)$.

On relie alors les gains optimaux des états comme suit:
$Opt(s) = \max_{a \in Act(s)} \{v(s,a) + Opt(T(s,a))\}.$

C'est une version très simplifiée de l'équation de Bellmann.

Programmation dynamique stochastique

$Opt(s) = \max_{a \in Act(s)} \{v(s,a) + Opt(T(s,a))\}.$

Implicitement on a supposé dans l'équation précédente que le système était déterministe, ie que l'impact d'une action n'était pas aléatoire.

On peut généraliser le raisonnement lorsqu'il y a intervention du hasard, ie lorsque l'impact d'une action n'est pas déterministe.

Exemple: le jeu du cochon

Le jeu du cochon est un jeu à deux joueurs qui se joue avec un dé classique; le gagnant est le premier qui a un score total d'au moins 100 (par exemple) points.

- Au départ, chacun a un score total de 0 point;
- Chaque jour joue à tour de rôle: après chaque lancer de dé, le joueur a deux possibilités: relancer le dé ou passer la main;
- Si il lance le dé:
  - si il obtient un 1, il ajoute 1 à son score total - quelque soit le score du tour- et passe la main;
  - si il fait un nombre de 2 à 6, ce nombre est ajouté au score du tour.
- Si le joueur décide de passer la main, il ajoute à son score le maximum du score du tour et de 1.

Exemple de déroulement

Au départ A et B ont tous les deux un score nul. A lance le dé; il fait un 6.

> Si il décide de passer la main, on arrive à B:0; A:6.
>
> Sinon il décide de continuer, le score du tour est 6.
>
> > Si il obtient un 1 à son deuxième lancer de dé, on arrive à B: 0 A:1
> > sinon si il obtient un 2 le score du tour est 8.
> >
> > > Si il décide de passer la main, on arrive à B:0; A: 8
> > > Si il décide de continuer .....

Modélisation: les configurations

Une configuration du jeu c’est

Le nom du joueur qui a le dé en main.

le score de chaque joueur

le score courant du joueur dont c’est le tour

Modélisation: les actions

A chaque état:

- soit l'état est gagnant
- soit non et le joueur dont c'est le tour peut
  - passer la main: l'action est alors déterministe
  - relancer le dé: il y a alors plusieurs configuratuons atteignables avec chacune une probabilité de $1/6$.

Modélisation: la valeur, l'équation

Valeur d'un état= probabilité qu'a celui dont c'est le tour de gagner. On a alors, si x est le score du joueur courant, y celui de son adversaire, $sctour$ le score courant du tour

$Val(A, x, y, sctour) = Val(B, x, y, sctour) = prob(x, y, sctour)$ avec:

$$\begin{aligned} prob(x, y, sctour) &= 1 \text{ si } x + sctour >= 100 \\ &= max(pP(x, y, sctour), pL(x, y, sctour)) \text{ sinon} \end{aligned}$$

$pP(A, x, y, sctour) = 1 - prob(B, y, x + sctour, 0)$
// le joueur passe la main

$pL(x, y, sctour) = 1/6*((1-prob(y, x+1, 0)+\Sigma_{i=2}^{6} prob(x, y, sctour+i))$
// cas où le joueur lance le dé

Le calcul

$prob(x, y, sctour) = 1$ si $x + sctour >= 100$
$= max(pP(x, y, sctour), pL(x, y, sctour))$ sinon

$pP(A, x, y, sctour) = 1 - prob(B, y, x + sctour, 0)$
// le joueur passe la main

$pL(x, y, sctour) = 1/6 * ((1 - prob(y, x+1, 0) + \Sigma_{i=2}^{6} prob(x, y, sctour + i)$
// cas où le joueur lance le dé

On est typiquement dans le cas de la programmation dynamique.

Question? Le graphe est bien acyclique?

Question ? dans quel ordre faire l'évaluation?

Le code: les tables

3 tables indexées par trois entiers de 0 à 100 correspondant aux trois scores:
Comp : table de booléens pour savoir ce qui a été calculé

Tprob: table de proba $Tprob[x][y][sctour]$ vaudra $prob(x, y, sctour)$

strat : table de stratégies $strat[x][y][sctour]$ vaudra *faux* si il vaut mieux ne pas lancer le dé, *vrai* sinon.

le code

```
double prob(int x, int y, int sctour){
  if (x+ sctour >=100) return 1;
  if (!comp[x][y][sctour]) {
    comp[x][y][sctour]=true;
    double pp=prob_passe(x,y,sctour);
    double pl=prob_lance(x,y,sctour);
    if (pp>=pl )
      {strat[x][y][sctour]=false;
      Tprob[x][y][sctour]= pp;}
    else
      {strat[x][y][sctour]=true;
      Tprob[x][y][sctour]= pl;}
  }
return Tprob[x][y][sctour];}
```

le code

```
double prob_lance(int x, int y, int sctour){
return (
1.0/6.0 * (1 -prob (y,x+1,0)
     + prob (x,y,sctour+2)
     + prob (x,y,sctour+3)
     + prob (x,y,sctour+4)
     + prob (x,y,sctour+5)
     + prob (x,y,sctour+6)));}


double prob_passe(int x, int y, int sctour){
return 1 -prob (y,x+sctour,0);}
```

D’un paradigme à l’autre: Programmation dynamique/ ”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.

D’un paradigme à l’autre: représentation arborescente

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).

Typiquement un algorithme glouton n’explore qu’une seule branche de l’arbre.