Introduction à la programmation dynamique

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

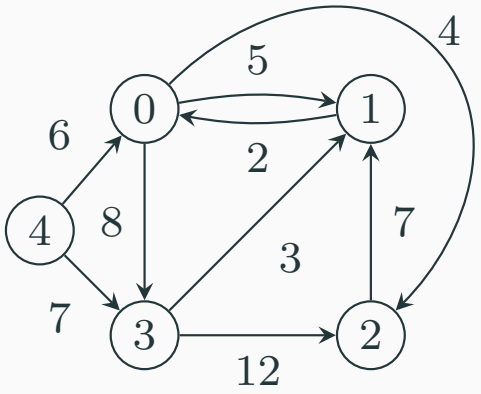
Introduction

Programmation dynamique

La **programmation dynamique** est une technique pour résoudre des problèmes d'**optimisation** : sur un univers $\mathcal{U}$, on cherche à **minimiser** (ou **maximiser**, la situation est symétrique) une certaine fonction, souvent à valeurs dans les entiers.

Concrètement, on se donne $f : \mathcal{U} \to \mathbb{Z}$, et on cherche à déterminer $x \in \mathcal{U}$ tel que $f(x) = \min_{y \in \mathcal{U}}\{f(y)\}$ ou $f(x) = \max_{y \in \mathcal{U}}\{f(y)\}$, si cette quantité existe.

Exemples



Exemple : plus court chemin dans un graphe

La figure ci-dessus représente un **graphe pondéré** : les **sommets** sont reliés par des **arêtes**, ces arêtes sont étiquetées par des **poids**. On peut se déplacer d'un sommet à un autre en suivant une arête, mais ce déplacement a un **coût** (correspondant au poids de l'arête).

Question : quel est le poids **minimal** d'un chemin entre le sommet 3 et le sommet 2 ?

Exemples

$$A = \begin{pmatrix} 2 & 39 & 12 & 49 & 47 & 18 & 22 & 19 \\ 37 & 21 & 34 & 26 & 10 & 2 & 35 & 39 \\ 31 & 21 & 12 & 26 & 34 & 27 & 7 & 22 \\ 20 & 46 & 16 & 2 & 11 & 40 & 36 & 13 \\ 18 & 30 & 32 & 37 & 28 & 24 & 9 & 6 \end{pmatrix}$$

Exemple

Dans la matrice ci-dessus, on s'intéresse au chemin de la case en haut à gauche vers la case en bas à droite, utilisant seulement les déplacements $\rightarrow$ et $\downarrow$, qui maximise la somme des entiers rencontrés sur le chemin.

Question : quel est ce chemin, et son poids ?

On va résoudre ce problème dans ce chapitre.

Exemples

Exemple

On appelle sous-séquence commune à deux chaînes de caractères s et t une chaîne x dont les caractères apparaissent dans l'ordre à la fois dans s et dans t (avec possiblement des caractères intercalés).

Question : quelle est la (une) plus longue sous-séquence commune à “arythmie” et “rhomboédrique” ?

Programmation dynamique

On verra que la programmation dynamique est une technique qui peut s'appliquer pour résoudre algorithmiquement ces problèmes de manière efficace.

Néanmoins, elle ne s'applique pas à tous les problèmes d'optimisation, et on verra que lorsqu'elle s'applique elle n'est pas forcément la plus efficace.

Une exemple complet : chemin de poids maximal dans une matrice

Problème du chemin de poids maximal dans une matrice

Problème

On reprend le problème évoqué dans l'introduction : trouver le chemin partant de la case en haut à gauche d'une matrice $A = (a_{i,j})_{0 \leq i < n, 0 \leq j < m}$ constituée d'entiers positifs, et aboutissant à la case en bas à droite en n'utilisant que les déplacement $\rightarrow$ et $\downarrow$ dont le **poids** (somme des entiers rencontrés) est maximal.

Problème du chemin de poids maximal dans une matrice

Recherche exhaustive

On pourrait essayer d'examiner tous les chemins possibles, car ils sont en nombre fini. Dénombrons ces chemins.

Pour construire un tel chemin, il suffit de savoir où placer les $n-1$ déplacements $\downarrow$ parmi les $n-1+m-1$ déplacements totaux. Ainsi, il y a $N_{n,m} = \binom{n+m-2}{n-1}$ chemins possibles.

Utilisons la formule de Stirling pour calculer un équivalent asymptotique de cette quantité lorsque $n = m$.

Problème du chemin de poids maximal dans une matrice

Recherche exhaustive

$$N_{n,n} = \binom{2n-2}{n-1} = \frac{(2n-2)!}{(n-1)!^2}$$

$$\underset{n\to\infty}{\sim} \frac{(2n-2)^{2n-2}e^{-2(n-1)}\sqrt{2\pi(2n-2)}}{2(n-1)^{2(n-1)}e^{-2(n-1)}\pi(n-1)}$$

$$= \frac{2^{2n-2}}{\sqrt{\pi(n-1)}} \underset{n\to\infty}{\sim} \frac{2^{2n-2}}{\sqrt{\pi n}}$$

Le nombre de chemins possibles est donc **exponentiel** en n. Pour $n \geq 30$, un algorithme qui explore tous les chemins est **inutilisable** en pratique.

Problème du chemin de poids maximal dans une matrice

Solution aux sous-problèmes

Considérons une solution à notre problème, c'est-à-dire une chemin c dans A de la case $(0,0)$ à la case $(n-1,m-1)$ dont le poids est maximal.

Supposons que ce chemin passe par la case (i,j).

Le chemin se décompose en deux morceaux

$$(0,0) \overset{c_1}{\rightsquigarrow} (i,j) \overset{c_2}{\rightsquigarrow} (n-1,m-1)$$

où c_1 et c_2 sont des chemins de poids maximal de sous-matrices de A (comme on se restreint aux déplacements $\rightarrow$ et $\downarrow$, c_1 et c_2 ne peuvent pas sortir de la sous-matrice en question).

Problème du chemin de poids maximal dans une matrice

Preuve

par l'absurde, supposons que c_1 n'est pas optimal. Il existe donc un chemin c_1' de poids strictement supérieur de la case $(0,0)$ à la case (i,j). Mais alors, le chemin c' suivant

$$(0,0) \overset{c_1'}{\rightsquigarrow} (i,j) \overset{c_2}{\rightsquigarrow} (n-1, m-1)$$

est un chemin de poids strictement supérieur à c : impossible.

De même pour c_2.

Problème du chemin de poids maximal dans une matrice

Solution aux sous-problèmes

Le problème d'optimisation d'un chemin de la case $(0, 0)$ à une case (i, j) peut être qualifié de **sous-problème** au problème initial, car la matrice à considérer est plus petite.

Une solution au problème initial (sur la matrice $n \times m$) donne donc une solution à de multiples sous-problèmes : c'est une caractéristique qui indique que la **programmation dynamique** peut être utilisée pour résoudre ce problème.

Une relation récursive pour le poids maximal d'un chemin

Une relation récursive

Notons $(p_{i,j})_{0 \leq i < n, 0 \leq j < m}$ le poids maximal d'un chemin de $(0,0)$ à (i,j). Résoudre le problème consiste à trouver un chemin de poids $p_{n-1,m-1}$ de $(0,0)$ à $(n-1, m-1)$.

Concentrons nous d'abord sur le calcul de $p_{n-1,m-1}$, et d'une manière générale de tous les $(p_{i,j})_{0 \leq i < n, 0 \leq j < m}$. Remarquons que les $(p_{i,j})$ satisfont la relation suivante :

$$p_{i,j} = a_{i,j} + \begin{cases} 0 & \text{si } i = j = 0 \\ p_{i,j-1} & \text{si } i = 0 \text{ et } j > 0 \\ p_{i-1,j} & \text{si } j = 0 \text{ et } i > 0 \\ \max\{p_{i-1,j}, p_{i,j-1}\} & \text{sinon} \end{cases}$$

Une relation récursive pour le poids maximal d'un chemin

Preuve

Dans les 3 premiers cas, la relation est évidente, car dans ce cas il n'y a qu'un seul chemin possible pour aller de $(0,0)$ à (i,j). Supposons maintenant $i > 0$ et $j > 0$.

- À partir d'un chemin c de $(0,0)$ à $(i-1,j)$, on en construit un allant jusqu'à (i,j) via un déplacement $\downarrow$. En choisissant c de poids maximal $p_{i-1,j}$, on obtient $p_{i,j} \geq a_{i,j} + p_{i-1,j}$. De manière analogue, on obtient $p_{i,j} \geq a_{i,j} + p_{i,j-1}$. Donc $p_{i,j} \geq a_{i,j} + \max\{p_{i-1,j}, p_{i,j-1}\}$.
- Réciproquement, tout chemin menant à la case (i,j) passe par $(i-1,j)$ ou $(i,j-1)$. Prenons-en un de poids maximal $p_{i,j}$ et supprimons le dernier mouvement. On obtient un chemin de poids $p_{i,j} - a_{i,j}$ menant à $(i-1,j)$ ou $(i,j-1)$. Ainsi, $\max\{p_{i-1,j}, p_{i,j-1}\} \geq p_{i,j} - a_{i,j}$.

Une relation récursive pour le poids maximal d'un chemin

Un calcul itératif des $p_{i,j}$

Cette relation fournit un algorithme récursif pour le calcul de $p_{n-1,m-1}$.

Néanmoins cet algorithme revient à explorer tous les chemins possibles et a la même complexité que la recherche exhaustive.

Il est toutefois possible de calculer tous les coefficients $p_{i,j}$ très simplement en utilisant une matrice P, de même taille que A, que l'on remplit en temps $O(mn)$ à l'aide des coefficients de A en suivant la relation précédente.

Un calcul itératif des $p_{i,j}$

$$p_{0,0} = a_{0,0} \ ; \ p_{i,j} = a_{i,j} + \begin{cases} p_{i,j-1} & \text{si } i = 0 \text{ et } j > 0 \\ p_{i-1,j} & \text{si } j = 0 \text{ et } i > 0 \\ \max\{p_{i-1,j}, p_{i,j-1}\} & \text{sinon} \end{cases}$$

code OCaml

```
let calcul_p a =
  let n,m = Array.length a, Array.length a.(0) in
  let p = Array.make_matrix n m a.(0).(0) in
  for i=1 to n-1 do
    p.(i).(0) <- p.(i-1).(0) + a.(i).(0)
  done ;
  for j=1 to m-1 do
    p.(0).(j) <- p.(0).(j-1) + a.(0).(j)
  done ;
  for i=1 to n-1 do
    for j=1 to m-1 do
      p.(i).(j) <- a.(i).(j) + max p.(i-1).(j) p.(i).(j-1)
    done
  done ;
  p
;;
```

Un calcul itératif des $p_{i,j}$

code C

```c
int max(int a, int b) {
  if (a>b) return a;
  else return b;
}

int **calcul_p(int n, int m, int a[n][m]) {
  int **p = (int**)malloc(sizeof(int*) * n);
  for (int i = 0; i < n; i++) {
    p[i] = malloc(sizeof(int) * m);
  }

  p[0][0] = a[0][0]; // initialisation de la première case

  for (int i = 1; i < n; i++) { // initialisation de la première colonne
    p[i][0] = a[i][0] + p[i-1][0];
  }

  for (int j = 1; j < m; j++) { // initialisation de la première ligne
    p[0][j] = a[0][j] + p[0][j-1];
  }

  for (int i = 1; i < n; i++) { // calcul itératif
    for (int j = 1; j < m; j++) {
      p[i][j] = a[i][j] + max(p[i-1][j],p[i][j-1]);
    }
  }
  return p;
}
```

Un calcul itératif des $p_{i,j}$

Remarque

Il faut bien faire attention à remplir les coefficients de la matrice dans un ordre adapté. Ici, pour calculer $p_{i,j}$, il faut avoir déjà rempli les cases $p_{i-1,j}$ et $p_{i,j-1}$ si elles existent.

Exemple

Sur la matrice A de l'introduction, on obtient :

$$P = \begin{pmatrix} 2 & 41 & 53 & 102 & 149 & 167 & 189 & 208 \\ 39 & 62 & 96 & 128 & 159 & 169 & 224 & 263 \\ 70 & 91 & 108 & 154 & 193 & 220 & 231 & 285 \\ 90 & 137 & 153 & 156 & 204 & 260 & 296 & 309 \\ 108 & 167 & 199 & 236 & 264 & 288 & 305 & 315 \end{pmatrix}$$

Détermination d'une solution au problème initial

Obtenir le chemin de poids maximal

On sait maintenant calculer les $p_{i,j}$ dans un temps acceptable, il reste à déterminer un chemin de poids $p_{n-1,m-1}$ dans la matrice A allant de $(0,0)$ à $(n-1,m-1)$.

Solution

Une solution consiste à remonter de la case $(n-1,m-1)$ à la case $(0,0)$ dans la matrice P. Depuis la case (i,j), on a le choix entre remonter à la case $(i-1,j)$ ou à la case $(i,j-1)$.

Il suffit de choisir la case $(i',j') \in \{(i-1,j),(i,j-1)\}$ telle que $p_{i',j'}$ est maximal : on obtient alors un chemin optimal avec une complexité supplémentaire en $O(m+n)$.

Détermination d'une solution au problème initial

code OCaml

```
let max_chemin a =
  let n,m = Array.length a, Array.length a.(0) in
  let p = calcul_p a in
  let rec remonte i j l = match (i,j) with
    | 0,0 -> l
    | 0,_ -> remonte 0 (j-1) (">"::l)
    | _,0 -> remonte (i-1) 0 ("v"::l)
    | _,_ when p.(i-1).(j) > p.(i).(j-1) -> remonte (i-1) j ("v"::l)
    | _,_ -> remonte i (j-1) (">"::l)
  in remonte (n-1) (m-1) []
;;
```

Implémentation

On encode un chemin par une liste de caractères “>” ou “v” indiquant à chaque étape un déplacement $\rightarrow$ ou $\downarrow$.

On remonte récursivement depuis la case $(n-1, m-1)$ en insérant les caractères dans une liste.

Détermination d'une solution au problème initial

code C

```c
char *max_chemin(int n, int m, int a[n][m]) {
  int **p = calcul_p(n,m,a); // Attention, on vient d'allouer de la mémoire !
  int i = n-1;
  int j = m-1;
  char *res = malloc(sizeof(char) * (i+j));
  while (i != 0 || j !=0) {
    if (i == 0) { res[i+j-1] = '>'; j--; }
    else if (j == 0) { res[i+j-1] = 'v'; i--; }
    else if (p[i-1][j] > p[i][j-1]) { res[i+j-1] = 'v'; i--; }
    else { res[i+j-1] = '>'; j--; }
  }

  /* Il ne faut pas oublier de libérer p et toutes ses lignes,
   * sinon on a une fuite mémoire ! */
  for (i = 0; i < n; i++) {
    free(p[i]);
  }
  free(p);

  return res;
}
```

Détermination d'une solution au problème initial

```
# max_chemin a ;;
- : string list = [">"; ">"; ">"; ">"; "v"; "v"; ">"; "v"; ">"; ">"; "v"]
```

Exemple

Voici ce que renvoie notre algorithme sur la matrice A donnée dans l'introduction.

Autrement dit, un chemin de poids maximal dans A est le suivant :

$$A = \begin{pmatrix} \mathbf{2} & \mathbf{39} & \mathbf{12} & \mathbf{49} & \mathbf{47} & 18 & 22 & 19 \\ 37 & 21 & 34 & 26 & \mathbf{10} & 2 & 35 & 39 \\ 31 & 21 & 12 & 26 & \mathbf{34} & \mathbf{27} & 7 & 22 \\ 20 & 46 & 16 & 2 & 11 & \mathbf{40} & \mathbf{36} & \mathbf{13} \\ 18 & 30 & 32 & 37 & 28 & 24 & 9 & \mathbf{6} \end{pmatrix}$$

Matrices en C

code C

```c
int **creer_matrice(int n, int m) {
  int **p = (int**)malloc(sizeof(int*) * n);
  for (int i = 0; i < n; i++) {
    p[i] = malloc(sizeof(int) * m);
  }
  return p;
}

void detruire_matrice(int **p, int n, int m) {
  for (i = 0; i < n; i++) {
    free(p[i]);
  }
  free(p);
}
```

Matrices en C

Dans les exemples qui vont suivre, on devra gérer la mémoire des matrices à la main (pour les programmes en C).

Voici deux fonctions allouant et libérant la mémoire d'une matrice de taille $n \times m$.

Principes de la programmation dynamique, et variantes

Principes de la programmation dynamique, et variantes

Programmation dynamique

L'exemple précédent est typique d'une résolution du problème par **programmation dynamique**.

Donnons un résumé de la démarche, dans une cadre plus abstrait.

Résolution de problème par programmation dynamique

Problème

On se donne donc une fonction $f : \mathcal{U} \to \mathbb{Z}$, et on cherche à déterminer x tel que $f(x) = \max_{y \in \mathcal{U}}\{f(y)\}$ (la démarche est la même si on cherche à minimiser f sur $\mathcal{U}$).

Dans la suite, on notera $M_{f,\mathcal{U}} = \max_{y \in \mathcal{U}}\{f(y)\}$.

Démarche

Il y a 4 étapes dans la résolution d'un tel problème par **programmation dynamique**.

Résolution de problème par programmation dynamique

Étape 1

Identifier une **sous-structure optimale** : c'est un indice que la programmation dynamique peut être appliquée.

Il s'agit de voir que si l'ont connaît $x \in \mathcal{U}$ tel que $f(x) = M_{f,\mathcal{U}}$, alors de x on déduit des solutions $(x_i)_{i \in I}$ à des sous-problèmes de la forme "trouver $x_i \in \mathcal{U}_i$ tel que $f_i(x_i) = M_{f_i,\mathcal{U}_i}$".

Les sous-problèmes doivent être plus simples à résoudre que le problème initial.

Exemple

Dans l'exemple précédent, les sous-problèmes concernaient des matrices plus petites.

Résolution de problème par programmation dynamique

Étape 2

Déduire de la sous-structure optimale une **relation récursive** permettant de calculer $M_{f,\mathcal{U}}$ à partir de certains $M_{f_i,\mathcal{U}_i}$.

Exemple

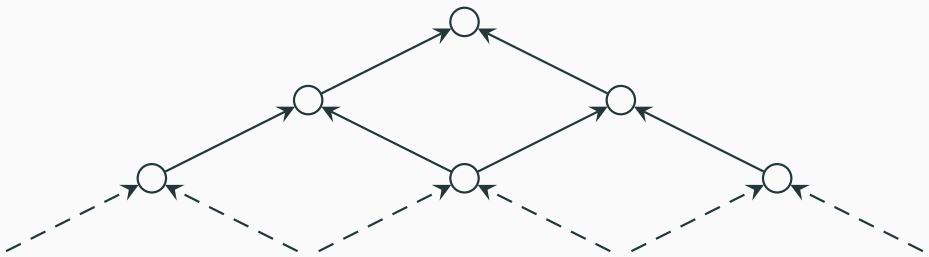
Dans l'exemple précédent, on a exhibé une relation entre $p_{n-1,m-1}$, $p_{n-2,m-1}$ et $p_{n-1,m-2}$.

Résolution de problème par programmation dynamique

Remarque

Ce qui distingue une résolution par **programmation dynamique** d'un algorithme "**diviser pour régner**" est le fait que les calculs des différents $M_{f_i,\mathcal{U}_i}$ ne sont pas du tout **indépendants**.

Écrire un algorithme récursif calculant naïvement $M_{f,\mathcal{U}}$ mène en général à une solution très couteuse car on effectuera des appels récursifs qui se **chevauchent**, comme on l'a vu sur l'exemple de la suite de Fibonacci.



Résolution de problème par programmation dynamique

Étape 3

Pour palier ce problème, on calcule les $M_{f_i,\mathcal{U}_i}$ utiles **itérativement**, en faisant usage d'un **tableau** pour stocker tous ces éléments.

On peut parfois se contenter de n'en stocker que certains.

Exemple

Dans l'exemple précédent, on pourrait ne stocker qu'une ligne de la matrice P, ce qui mènerait à une complexité spatiale en $O(n)$ au lieu de $O(n \times m)$.

Pour la suite de Fibonacci, on n'a besoin de stocker uniquement les deux dernière valeurs, ce qui permet d'avoir une complexité spatiale en $O(1)$ (au lieu de $O(n)$).

Résolution de problème par programmation dynamique

Étape 4

Enfin, on modifie légèrement le calcul des $M_{f_i,\mathcal{U}_i}$ pour obtenir en même temps $\forall i \in I$ un x_i satisfaisant $f_i(x_i) = M_{f_i,\mathcal{U}_i}$.

En général, cette étape n'est pas difficile.

Exemple

Dans l'exemple précédent, on a choisi de ne calculer un chemin optimal qu'après le calcul des $p_{i,j}$, mais on aurait pu par exemple stocker en parallèle des $p_{i,j}$ (dans une autre matrice) le dernier déplacement à effectuer pour arriver en (i, j) en suivant un chemin optimal.

Une parenthèse sur les problèmes de combinatoire

Problèmes de combinatoire

La technique qu'on vient de voir pour résoudre un problème d'optimisation s'applique aussi pour la résolution de certains problèmes de combinatoire.

Dans ce cas, on cherche plutôt à calculer la taille d'un certain ensemble $\mathcal{U}$.

Une parenthèse sur les problèmes de combinatoire

Problèmes de combinatoire

La démarche ressemble à celle vue précédemment :

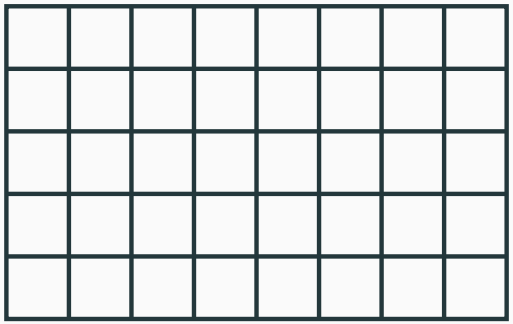
1. On partitionne $\mathcal{U}$ en sous-ensembles disjoints $(\widetilde{\mathcal{U}}_i)_i$, les $(\mathcal{U}_i)$ étant des ensembles associés à des "sous-problèmes combinatoires" et les $(\widetilde{\mathcal{U}}_i)$ des ensembles en bijection avec les $(\mathcal{U}_i)$: une légère modification d'un élément de $\mathcal{U}_i$ donne un élément de $\widetilde{\mathcal{U}}_i$.
2. Écrire que $\mathcal{U} = \sqcup_i \widetilde{\mathcal{U}}_i$ fournit une relation de récurrence permettant de calculer $|\mathcal{U}|$, à savoir $|\mathcal{U}| = \sum_i |\mathcal{U}_i|$.
3. On calcule plutôt $|\mathcal{U}|$ itérativement, en faisant usage d'un tableau.

Une parenthèse sur les problèmes de combinatoire

Problèmes de combinatoire

Pour résumer, la résolution d'un problème de combinatoire suit essentiellement les étapes 1 à 3 évoquées pour la résolution d'un problème d'optimisation par programmation dynamique, l'étape 4 n'ayant pas de sens ici.

Exemple : nombre de chemins dans un quadrillage



Exemple

On considère un quadrillage de taille $n \times m$ comme ci-dessus.

On cherche le nombre de chemins partant du coin en haut à gauche $(0, 0)$ jusqu'au coin en bas à droite (n, m), en suivant les directions $\rightarrow$ et $\downarrow$.

On a déjà vu que ça vaut $\binom{n+m}{m}$, mais appliquons la méthode qu'on vient d'évoquer.

Exemple : nombre de chemins dans un quadrillage

Exemple

1. Notons $\mathcal{C}_{i,j}$ l'ensemble des chemins allant de $(0,0)$ à (i,j). Alors $\forall i,j \geq 0$, on a $\mathcal{C}_{i,j} = \widetilde{\mathcal{C}}_{i-1,j} \sqcup \widetilde{\mathcal{C}}_{i,j-1}$, où $\widetilde{\mathcal{C}}_{i-1,j}$ est l'ensemble des chemins de $\mathcal{C}_{i-1,j}$ complétés par le mouvement $(i-1,j) \to (i,j)$, et de même pour $\mathcal{C}_{i,j-1}$. Par convention, on pose $\mathcal{C}_{-1,j} = \mathcal{C}_{i,-1} = \emptyset$, et $\mathcal{C}_{0,0}$ contient un unique chemin.
2. En notant $N_{i,j} = |\mathcal{C}_{i,j}|$, on a donc $N_{i,j} = N_{i-1,j} + N_{i,j-1}$ pour $i,j > 0$, et $N_{i,0} = N_{0,j} = 1$.
3. On peut donc calculer les $N_{i,j}$ à l'aide d'un tableau de taille $(n+1) \times (m+1)$ pour obtenir au final $N_{n,m}$.

Exemple : nombre de chemins dans un quadrillage

Exemple

En fait, on a retrouvé indirectement la formule du triangle de Pascal.

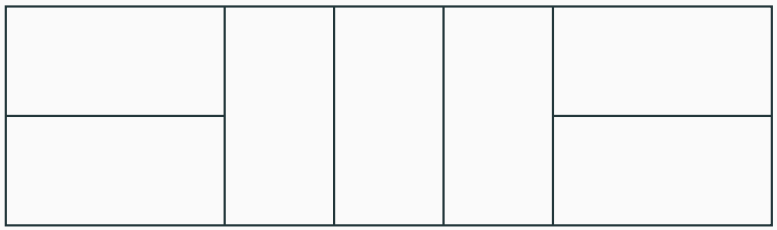
En implémentant cette méthode, on obtiendra bien $1287 = \binom{5+8}{5}$ pour la grille de tout à l'heure (de taille 5×8).

Exemple : un problème de pavage

Exemple

On considère un rectangle de taille $2 \times n$, et on s'intéresse aux **pavages** de ce rectangle par des dominos de taille 1×2.

Voici deux exemples de pavage d'un rectangle 2×7.



Exemple : un problème de pavage

Exemple

Notons F_n le nombre de pavages possibles d'un rectangle $2 \times n$.

Supposons $n \geq 2$ et considérons le domino occupant le coin en haut à gauche du rectangle.

- Si ce domino est placé verticalement (comme sur l'exemple de gauche), alors il reste à paver un rectangle $2 \times (n-1)$: donc F_{n-1} possibilités.
- Sinon, le domino est placé horizontalement (comme sur l'exemple de droite). Alors, un autre domino horizontal est placé nécessairement en dessous, et il reste à paver un rectangle $2 \times (n-2)$: donc F_{n-2} possibilités.

Exemple : un problème de pavage

Exemple

Ainsi, on a $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$ et $F_0 = F_1 = 1$.

On reconnait la suite de Fibonacci.

Comme on l'a vu dans un chapitre précédent, calculer cette suite par récurrence sera très inefficace.

Il vaut mieux utiliser un tableau pour se souvenir des valeurs précédentes et éviter d'avoir des appels récursifs qui se chevauchent.

En fait, il suffit de se souvenir uniquement des deux dernières valeurs de la suite.

Exemple : un problème de pavage

Remarque

Ce problème était facile. Voici un problème moins évident : paver un rectangle 3×30 avec des dominos 1×2.

Algorithmes “glouton”

Retour sur le problème du chemin maximal

Revenons au problème de trouver un chemin de poids maximal dans une matrice, qui n'utilise que les directions $\rightarrow$ et $\downarrow$.

On a vu que le poids maximal d'un chemin de $(0,0)$ à (i,j) vérifiait : $p_{i,j} = a_{i,j} + \max\{p_{i-1,j}, p_{i,j-1}\}$.

Au lieu de calculer tous les $p_{i,j}$, faisons un choix **localement optimal** en allant sur la case $(i',j') \in \{(i-1,j),(i,j-1)\}$ tel que $a_{i',j'}$ est maximal.

En faisant systématiquement ce choix (qu'on appelle le choix **glouton**) à chaque étape depuis $(n-1,m-1)$ vers $(0,0)$, on construit directement un unique chemin.

Algorithmes “glouton”

Exemple

Voici les chemins trouvés grâce à la **programmation dynamique** (de poids 315) et avec l'algorithme glouton (de poids 304).

L'algorithme glouton ne trouve ainsi pas le chemin optimal, mais pas loin.

$$A = \begin{pmatrix} \mathbf{2} & \mathbf{39} & \mathbf{12} & \mathbf{49} & \mathbf{47} & 18 & 22 & 19 \\ 37 & 21 & 34 & 26 & \mathbf{10} & 2 & 35 & 39 \\ 31 & 21 & 12 & 26 & \mathbf{34} & \mathbf{27} & 7 & 22 \\ 20 & 46 & 16 & 2 & 11 & \mathbf{40} & \mathbf{36} & \mathbf{13} \\ 18 & 30 & 32 & 37 & 28 & 24 & 9 & \mathbf{6} \end{pmatrix}$$

Algorithmes “glouton”

Principe des algorithmes glouton

Un **algorithme glouton** pour résoudre un problème d'optimisation suit les mêmes principes que la résolution par **programmation dynamique**, néanmoins l'étape 3 diffère.

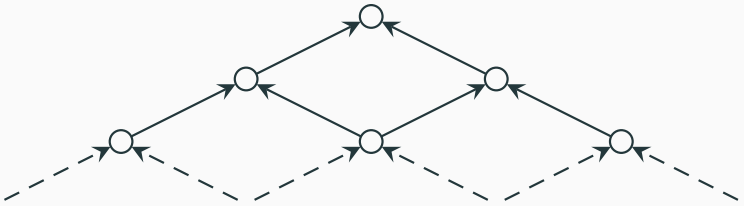
Au lieu d'utiliser un tableau pour calculer successivement tous les $M_{f_i,\mathcal{U}_i}$ (qui correspondent aux $p_{i,j}$ dans le problème du chemin maximal dans une matrice), l'algorithme glouton fait un choix local pour ramener le problème à un problème plus simple.

Le choix effectué est **localement optimal** (par exemple se diriger vers la case voisine ayant la plus grande valeur).

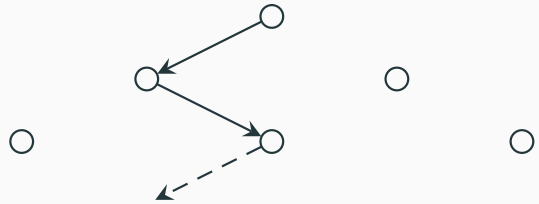
Algorithmes “glouton”

Principe des algorithmes glouton

Visuellement, on peut représenter les choix faits par un **algorithme glouton** par rapport à la **programmation dynamique** par le schéma suivant.



Programmation dynamique



Algorithme glouton

Algorithmes “glouton”

Utilisation des algorithmes glouton

On a vu que dans le problème du chemin de poids maximal dans une matrice, l'algorithme glouton ne donnait pas une réponse optimale.

Néanmoins, c'est le cas pour certains problèmes : il faut alors prouver que le choix **localement optimal** se révèle être un choix **globalement optimal**.

Exemple

Nous verrons un algorithme de calcul de plus courts chemins depuis une origine fixée dans un graphe pondéré à poids positifs, qui se révèle être un **algorithme glouton**.

Algorithmes “glouton”

Avantage des algorithmes glouton

L'intérêt d'un algorithme glouton par rapport à un algorithme faisant usage de programmation dynamique est sa **complexité**, en général bien moins élevée.

Exemple

Par exemple pour le problème précédent, l'algorithme glouton n'a qu'une complexité $O(n + m)$.

Deux autres exemples de résolution par programmation dynamique

Le problème de la plus longue sous-séquence commune

Plus longue sous-séquence commune

Ce problème, déjà évoqué dans l'introduction, consiste à trouver un mot x qui est une **sous-séquence commune maximale** à deux mots s et t.

Exemple

Par exemple, une sous-séquence commune maximale à "arythmie" et "rhomboédrique" est "rhmie", de longueur 5.

Le problème de la plus longue sous-séquence commune

Applications

Ce problème a des applications pratiques, notamment en génétique : la proximité de deux individus peut être évaluée en calculant la sous-séquence commune à deux séquences d'ADN prises sur les individus.

Remarque

La **distance d'édition** entre deux séquences d'ADN est également intéressante, et se calcule également par programmation dynamique.

Le problème de la plus longue sous-séquence commune

Sous-structure optimale

Notons n et m les longueurs de s et t.

Pour $0 \leq i \leq n$ et $0 \leq j \leq m$, on note $l_{i,j}$ la longueur d'une plus longue sous-séquence commune aux préfixes de tailles i et j de s et t.

Ce qui nous intéresse est $l_{n,m}$, et une sous-séquence associée.

Le problème de la plus longue sous-séquence commune

Sous-structure optimale

Supposons que l'on connaisse une sous-séquence commune x de longueur $l_{n,m}$ supposé strictement positif.

- Si les derniers caractères de s et t sont les mêmes, alors x termine par ce caractère (sinon, on pourrait le rajouter). Mais alors, x privé de son dernier caractère est une sous-séquence commune à s et t tous deux privés de leur dernier caractère (qu'on note s' et t'), et c'est même une plus longue sous-séquence commune de s' et t'.
  En effet, si ce n'était pas le cas, on pourrait trouver une sous-séquence commune à s et t plus longue que x en rajoutant le dernier caractère commun à s et t à une sous-séquence commune maximale de s' et t' : absurde.

Le problème de la plus longue sous-séquence commune

Sous-structure optimale

Supposons que l'on connaisse une sous-séquence commune x de longueur $l_{n,m}$ supposé strictement positif.

- Si les derniers caractères de s et t diffèrent, alors x est une sous-séquence commune à s et t' ou à s' et t (voire aux deux), et c'est même une plus longue sous-séquence commune de manière évidente.

Nous avons exhibé une sous-structure optimale.

Le problème de la plus longue sous-séquence commune

Relation de récurrence

La discussion précédente nous fournit une relation de récurrence sur les $l_{i,j}$:

$$l_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ 1 + l_{i-1,j-1} & \text{si } s.[i] = t.[j] \\ \max\{l_{i-1,j}, l_{i,j-1}\} & \text{sinon} \end{cases}$$

Calcul itératif des $l_{i,j}$

Pour calculer les $(l_{i,j})$, on procède itérativement en remplissant un tableau de taille $(n+1) \times (m+1)$.

Le problème de la plus longue sous-séquence commune

Construction d'une sous-séquence commune

De manière similaire au problème du chemin de poids maximal, il suffit de remonter depuis la case (n, m) jusqu'à la case $(0, 0)$ pour construire une plus longue sous-séquence commune.

Une fois les $l_{i,j}$ calculés, on connait la longueur d'une telle sous-séquence. On crée alors une chaîne de la bonne taille, qu'on va modifier. Pour cela, on remonte depuis la case (n, m).

- Si $l_{i,j} = l_{i-1,j}$ ou $l_{i,j-1}$, on peut remonter d'un cran vers le haut ou vers la gauche.
- Sinon, on a trouvé un nouveau caractère, et on remonte en diagonale à la case $(i - 1, j - 1)$.

Le problème de la plus longue sous-séquence commune

code C

```c
char *plssc(char *s, char *t) {
  int n = strlen(s), m = strlen(t);
  int **l = creer_matrice(n+1, m+1); // Allocation dynamique

  for (int i = 0; i <= n; i++) { l[i][0] = 0; } // Initialisation de la première colonne
  for (int j = 0; j <= m; j++) { l[0][j] = 0; } // Initialisation de la première ligne
  for (int i = 1; i <= n; i++) { // Calcul itératif de l
    for (int j = 1; j <= m; j++) {
      if (s[i-1] == t[j-1]) { l[i][j] = 1 + l[i-1][j-1]; }
      else { l[i][j] = max(l[i-1][j],l[i][j-1]); }
    }
  }
  char *x = (char*)malloc(sizeof(char) * l[n][m]);
  int i = n, j = m, k = l[n][m] -1;
  while (k >= 0) {
    if (l[i][j] == l[i-1][j]) { i--; }
    else if (l[i][j] == l[i][j-1]) { j--; }
    else {
      x[k] = s[i-1];
      i--;
      j--;
      k--;
    }
  }
  detruire_matrice(l, n+1, m+1); // Libération de la mémoire

  return x;
}
```

Le problème de la plus longue sous-séquence commune

Remarque

Depuis la version 4.02 d'OCaml, on ne peut plus modifier directement une chaîne de caractères.

Mais les chaînes de caractères OCaml utilisant le code **ASCII** (qui encode chaque caractère sur un octet), on peut s'en sortir avec la fonction `Bytes.set` ("**byte**" signifie "**octet**" en anglais).

Ainsi, on peut modifier le i-ème caractère d'une chaîne s par un caractère c via l'instruction suivante.

```
1  Bytes.set s i c
```

Le problème de la plus longue sous-séquence commune

code OCaml

```
let plssc s t =
  let n, m = String.length s, String.length t in
  let l = Array.make_matrix (n+1) (m+1) 0 in
  for i=1 to n do
    for j=1 to m do
      if s.[i-1]=t.[j-1] then
        l.(i).(j) <- 1+l.(i-1).(j-1)
      else
        l.(i).(j) <- max l.(i-1).(j) l.(i).(j-1)
    done
  done ;
  let x = String.make l.(n).(m) 'a' in
  let rec remonte i j k =
    (* on est sur la case (i,j) de l, et on cherche la k-ème lettre de x *)
    match k with
    | -1 -> ()
    | _ when l.(i).(j) = l.(i-1).(j) -> remonte (i-1) j k
    | _ when l.(i).(j) = l.(i).(j-1) -> remonte i (j-1) k
    | _ -> Bytes.set x k s.[i-1] ; remonte (i-1) (j-1) (k-1)
  in
  remonte n m (l.(n).(m)-1) ;
  x
;;
```

Le problème de la plus longue sous-séquence commune

Exemple

```
# plssc "arythmie" "rhomboedrique" ;;
- : string = "rhmie"
```

Complexité

Calculer les $l_{i,j}$ se fait en temps $O(nm)$.

Construire la plus longue sous-séquence commune se fait en $O(n+m)$.

Plus grand carré de zéros dans une matrice binaire

Plus grand carré de zéros dans une matrice binaire

Pour ce dernier exemple, on se donne une matrice $A = (a_{i,j})$ de taille $n \times m$, constituée de 0 et de 1.

On cherche la taille du plus grand carré de zéros dans cette matrice.

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Plus grand carré de zéros dans une matrice binaire

Plus grand carré de zéros dans une matrice binaire

Pour ce dernier exemple, on se donne une matrice $A = (a_{i,j})$ de taille $n \times m$, constituée de 0 et de 1.

On cherche la taille du plus grand carré de zéros dans cette matrice.

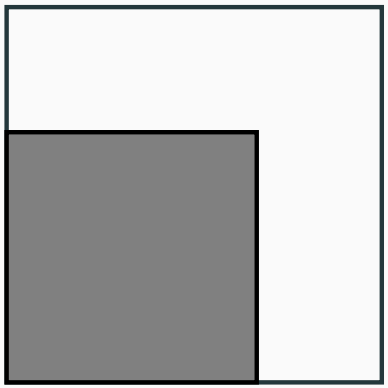
$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

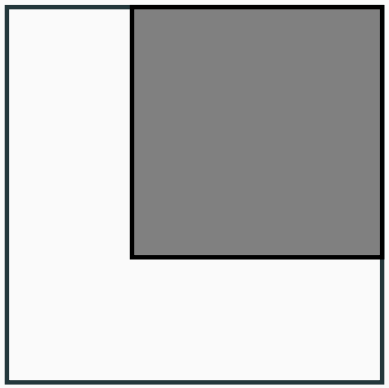
Plus grand carré de zéros dans une matrice binaire

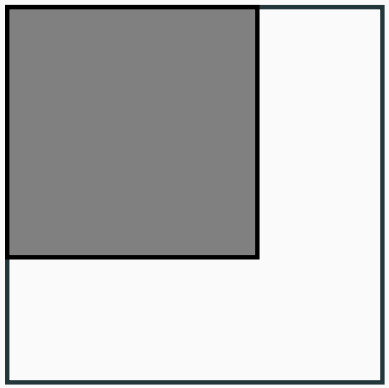
Sous-structure optimale

Si on sait qu'un carré de zéros de taille $p > 0$ a son coin en bas à droite à l'indice (i, j), cela signifie que les carrés de taille $p-1$ dont le coin en bas à droite est parmi les valeurs suivantes sont tous remplis de zéros :

$$\{(i-1, j), (i, j-1), (i-1, j-1)\}$$







Plus grand carré de zéros dans une matrice binaire

Relation de récurrence

La découverte de la sous-structure optimale nous permet d'exhiber facilement la relation de récurrence suivante, sur la taille $t_{i,j}$ du plus grand carré de zéros terminant à l'indice (i, j) (avec pour convention que $t_{-1,j} = t_{i,-1} = 0$).

$$t_{i,j} = \begin{cases} 0 & \text{si } a_{i,j} = 1 \\ 1 + \min\{t_{i-1,j}, t_{i,j-1}, t_{i-1,j-1}\} & \text{sinon} \end{cases}$$

Plus grand carré de zéros dans une matrice binaire

Construction du plus grand carré de zéros

On peut donc utiliser un tableau pour calculer itérativement les valeurs $t_{i,j}$.

Durant ce calcul, on peut stocker dans des références la valeur maximale de $t_{i,j}$ ainsi que les indices (i, j) associés.

Plus grand carré de zéros dans une matrice binaire

code C

```c
int min2(int a, int b) {
  if (a<b) return a;
  else return b;
}

int min3(int a, int b, int c) {
  return min2(a, min2(b,c));
}

struct result {
  int size;
  int i;
  int j;
};
typedef struct result result;
```

Plus grand carré de zéros dans une matrice binaire

code C

```c
result pgcz(int n, int m, int a[n][m]){
  result res = {0, 0, 0};

  int **t = creer_matrice(n, m); // Allocation dynamique

  for (int i = 0; i < n; i++) { // Initialisation de la première colonne
    if (a[i][0] == 0) { t[i][0] = 1; }
    else { t[i][0] = 0; }
    if (t[i][0] > res.size) { res.size = t[i][0]; res.i = i; res.j = 0; }
  }

  for (int j = 0; j < m; j++) { // Initialisation de la première ligne
    if (a[0][j] == 0) { t[0][j] = 1; }
    else { t[0][j] = 0; }
    if (t[0][j] > res.size) { res.size = t[0][j]; res.i = 0; res.j = j; }
  }

  for (int i = 1; i < n; i++) { // Calcul itératif
    for (int j = 1; j < m; j++) {
      if (a[i][j] == 0) { t[i][j] = 1 + min3(t[i-1][j], t[i][j-1], t[i-1][j-1]); }
      if (t[i][j] > res.size) { res.size = t[i][j]; res.i = i; res.j = j; }
    }
  }

  detruire_matrice(t, n, m); // Libération de la mémoire

  return res;
}
```

Plus grand carré de zéros dans une matrice binaire

code OCaml

```
let pgcz a =
  let n,m = Array.length a, Array.length a.(0) in
  let t = Array.make_matrix n m 0 in
  let p = ref 0 and ip = ref 0 and jp = ref 0 in (* valeur maximale de t et indices associés *)
  for i=0 to (n-1) do (* calcul de la première ligne *)
    if a.(i).(0) = 0 then t.(i).(0) <- 1 ;
    if t.(i).(0) > !p then
      begin
        p := t.(i).(0) ; ip := i ; jp := 0
      end
  done ;
  for j=0 to (m-1) do (* calcul de la première colonne *)
    if a.(0).(j) = 0 then t.(0).(j) <- 1 ;
    if t.(0).(j) > !p then
      begin
        p := t.(0).(j) ; ip := 0 ; jp := j
      end
  done ;
  for i=1 to (n-1) do (* calcul du reste du tableau *)
    for j=1 to (m-1) do
      if a.(i).(j) = 0 then t.(i).(j) <- 1 + min t.(i-1).(j) (min t.(i).(j-1) t.(i-1).(j-1)) ;
      if t.(i).(j) > !p then
        begin
          p := t.(i).(j) ; ip := i ; jp := j
        end
    done ;
  done ;
  (!p,!ip,!jp) ;;
```

Plus grand carré de zéros dans une matrice binaire

Exemple

```
# let a =
  [| [| 1 ; 0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1 ; 0 |] ;
     [| 0 ; 1 ; 1 ; 1 ; 0 ; 1 ; 1 ; 0 ; 0 |] ;
     [| 0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0 ; 0 |] ;
     [| 0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0 ; 0 |] ;
     [| 0 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 ; 0 |] ;
     [| 1 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 1 ; 0 |] |]
in pgcz a ;;
- : int * int * int = (3, 4, 8)
```