PROGRAMMATION DYNAMIQUE

La *programmation dynamique* est une méthode algorithmique utilisée pour résoudre certains problèmes d'*optimisation*, dans lesquels on cherche à minimiser ou maximiser une certaine fonction sur un univers $\mathcal{U}$.

1 Chemin de poids minimal dans une matrice

Étant donnée une matrice M d'entiers de taille $n \times p$, on cherche à déterminer le chemin de somme minimale allant du coin supérieur gauche au coin inférieur droit, en se déplaçant uniquement vers le bas ou vers la droite.

Par exemple, sur la matrice 5×5 suivante, le chemin de somme minimale est représenté en gras. Sa somme vaut 2427 :

$$\begin{pmatrix} \mathbf{131} & 673 & 234 & 103 & 18 \\ \mathbf{201} & \mathbf{96} & \mathbf{342} & 965 & 150 \\ 630 & 803 & \mathbf{746} & \mathbf{422} & 111 \\ 537 & 699 & 497 & \mathbf{121} & 956 \\ 805 & 732 & 524 & \mathbf{37} & \mathbf{331} \end{pmatrix}$$

On cherche à écrire une fonction calculant la somme d'un chemin minimal. On représentera en OCaml une matrice sous la forme d'un tableau de tableaux.

1.1 Relations de récurrence

Soit $M = (m_{ij})_{(i,j)\in[\![0,n[\![\times 0,p[\![} \in \mathcal{M}_{n,p}(\mathbb{R})$.

Pour tout couple d'entiers (i,j) tel $0 \leqslant i < n$ et $0 \leqslant j < p$, notons $s_{min}(i,j)$ la somme minimale d'un chemin allant de la case $(0,0)$ à la case (i,j) en se déplaçant uniquement vers la droite ou vers le bas.

Si un chemin de $(0,0)$ à (i,j) est de somme minimale, alors tout sous-chemin de ce chemin est aussi de somme minimale.

On peut affirmer que

- L'unique chemin entre $(0,0)$ et $(0,0)$ ne comporte que la case $(0,0)$, donc
$$s_{min}(0,0) = m_{00}$$
- Pour tout $j \in [\![1,p[\![$, l'avant dernière case du chemin de somme minimale de $(0,0)$ à $(0,j)$ est $(0,j-1)$, car les déplacements ne peuvent pas se faire vers la gauche, donc
$$s_{min}(0,j) = s_{min}(0,j-1) + m_{0j}$$
- Pour tout $i \in [\![1,n[\![$, l'avant dernière case du chemin de somme minimale de $(0,0)$ à $(i,0)$ est $(i-1,0)$, car les déplacements ne peuvent pas se faire vers le haut, donc
$$s_{min}(i,0) = s_{min}(i-1,0) + m_{i0}$$
- Pour $i \in [\![1,n[\![$ et $j \in [\![1,p[\![$, l'avant dernière case du chemin de somme minimale de $(0,0)$ à $(i,0)$ est soit $(i-1,j)$, soit $(i,j-1)$, et ce chemin privé de la case (i,j) est aussi de somme minimale, donc
$$s_{min}(i,j) = \min\{s_{min}(i-1,j), s_{min}(i,j-1)\} + m_{ij}$$

1.2 Algorithme naïf et complexité

```
let rec smin m i j =
  match (i, j) with
  | (0, 0) -> m.(0).(0)
  | (0, _) -> smin m 0 (j-1) + m.(0).(j)
  | (_, 0) -> smin m (i-1) 0 + m.(i).(0)
  | _ -> min (smin m (i-1) j) (smin m i (j-1)) + m.(i).(j)
;;
```

Pour construire un exemple avec lequel tester notre fonction, on peut utiliser la fonction `Array.make_matrix` :

```
let n, p = 15, 15;;
let m = Array.make_matrix n p 0;;
for i = 0 to n-1 do
  for j = 0 to p-1 do
    m.(i).(j) <- (i+1)*(2*j+1)
  done
done;;
```

On appelle ensuite la fonction `smin`, et on patiente un peu :

```
# smin m 14 14;;
- : int = 3480
```

Évaluons la complexité de la fonction `smin` : à chaque appel, on effectue d'éventuels appels récursifs auxquels ne s'ajoutent que des opérations en temps constants, donc le temps de calcul de $s_{min}(i,j)$ est un $O(C(i,j))$ où $C(i,j)$ est le nombre total d'appels à `smin` effectués lors de l'évaluation de `smin m i j`, et qui vérifie donc pour i et j non nuls :

$$\begin{aligned} C(0,0) &= 1 \\ C(i,0) &= 1 + C(i-1,0) \\ C(0,j) &= 1 + C(0,j-1) \\ C(i,j) &= 1 + C(i-1,j) + C(i,j-1) \end{aligned}$$

On montre alors que pour tout $(a,b) \in \mathbb{N}^2$, $C(a,b) = \binom{a+b+2}{b+1} - 1$.

Notamment, en utilisant la formule de Stirling, on obtient l'équivalent

$$C(n,n) \underset{n\to+\infty}{\sim} \frac{4^{n+1}}{\sqrt{\pi n}}$$

Par exemple, pour $n = 30$, $\frac{4^{n+1}}{\sqrt{\pi n}} \approx 4,75.10^{18}$, et $4,75.10^{18}$ ns ≈ 15 ans.

1.3 Mémoïsation

L'augmentation très rapide du temps de calcul de la fonction `smin` peut s'expliquer par le fait qu'elle calcule de nombreuses fois la même quantité : par exemple, pour i et j non nuls, le calcul de $s_{min}(i,j)$ conduit à calculer $s_{min}(i-1,j)$ et $s_{min}(i,j-1)$, qui conduisent chacun à calculer $s_{min}(i-1,j-1)$.

Par conséquent, le calcul de $s_{min}(n,n)$ nécessite (entre autres) deux fois le calcul de $s_{min}(n-1,n-1)$, qui nécessite lui-même deux fois le calcul de $s_{min}(n-2,n-2)$, etc.

Pour éviter de recalculer plusieurs fois la même quantité, la technique de *mémoïsation* consiste à stocker les valeurs s_{min} déjà calculées de dans une table ; on dit qu'on *tabule* les valeurs de la fonction s_{min}.

Pour cela, on ajoute à la fonction récursive `smin` un argument t correspondant à une table contenant les valeurs déjà calculées, ce qui permettra, avant tout calcul, de regarder si le résultat n'est pas déjà présent dans t. Cette table t sera remplie par effet de bord.

Ici, t sera une matrice de même taille que M, et telle que $t.(i).(j)$ contient soit la valeur -1, soit $s_{min}(i,j)$.

```
let rec smin t m i j =
  if t.(i).(j) >= 0
  then t.(i).(j)
  else
    let res = match (i,j) with
      | (0, 0) -> m.(0).(0)
      | (0, _) -> smin t m 0 (j-1) + m.(0).(j)
      | (_, 0) -> smin t m (i-1) 0 + m.(i).(0)
      | _ -> min (smin t m (i-1) j) (smin t m i (j-1)) + m.(i).(j)
    in
    t.(i).(j) <- res;
    res
;;
let t = Array.make_matrix n p (-1);;

smin t m (n-1) (p-1);;
```

1.4 Calcul itératif

Plutôt que de construire t par effet de bord, on peut remarquer de remplir t itérativement, en calculant les s_{min} de proche en proche en itérant par exemple sur i puis sur j.

```
let smin_it m =
  let n = Array.length m in
  let p= Array.length m.(0) in
  let t = Array.make_matrix n p (-1) in
  for i = 0 to n-1 do
    for j = 0 to p-1 do
      let res =
        match (i,j) with
        | (0,0) -> m.(0).(0)
        | (0, _) -> t.(0).(j-1) + m.(0).(j)
        | (_, 0) -> t.(i-1).(0) + m.(i).(0)
        | _ -> min t.(i-1).(j) t.(i).(j-1) + m.(i).(j)
      in
```

```
      t.(i).(j) <- res
    done
  done;
  t
;;
```

1.5 Complexité

Exercice 1 *Évaluer la complexité temporelle du calcul de $s_{min}(i,j)$ avec ces deux algorithmes.*

Exercice 2 *Évaluer la complexité spatiale du calcul de $s_{min}(i,j)$ avec ces deux algorithmes.*

Il est en fait possible de réduire la complexité spatiale en remarquant qu'il est uniquement nécessaire de calculer la dernière ligne de t, et que chaque ligne peut se déduire de la ligne précédente.

On peut donc calculer les lignes de proche en proche, en conservant au plus deux lignes en mémoire simultanément.

Ici, comme $t.(i-1).(j)$ n'est pas utilisé pour calculer $t.(i).(j')$ pour $j' > j$, il est même possible de ne garder que deux morceaux de lignes à un instant donné.

```
let smin_it2 m =
  let n = Array.length m in
  let p= Array.length m.(0) in
  let t = Array.make p (-1) in
  for i = 0 to n-1 do
    for j = 0 to p-1 do
      let res =
        match (i,j) with
        | (0,0) -> m.(0).(0)
        | (0, _) -> t.(j-1) + m.(0).(j)
        | (_, 0) -> t.(0) + m.(i).(0)
        | _ -> min t.(j) t.(j-1) + m.(i).(j)
      in
      t.(j) <- res
    done
  done;
  t.(p-1)
;;
```

Les invariants de boucle suivants sont en effet vérifiés :

- Á la fin de chaque itération de la boucle d'indice j, t contient $s_{min}(i,0)$, ..., $s_{min}(i,j)$ puis $s_{min}(i-1,j+1)$, ..., $s_{min}(i-1,p-1)$;
- Á la fin de chaque itération de la boucle d'indice i, t contient $s_{min}(i,0)$, ..., $s_{min}(i,p-1)$;

avec la convention : $s_{min}(-1,j) = -1$.

1.6 Obtention d'un chemin minimal

Exercice 3 *Que deviendrait la complexité de* `smin_it2` *si on stockait les chemins de sommes minimales sous forme de listes dans t plutôt que leurs sommes ?*

On peut créer de la redondance et stocker à la fois la somme minimale $s_{min}(i,j)$ et un chemin correspondant. Pour limiter la complexité spatiale *(quelle serait-elle ?)*, il n'est pas nécessaire de stocker l'intégralité de chaque chemin : on peut se contenter de garder en mémoire la case depuis laquelle on est arrivé sur la case (i,j), ce qui permet ensuite de retrouver le chemin de somme minimale.

```
let smin_it3 m =
  let n = Array.length m in
  let p= Array.length m.(0) in
  (* tableau des sommes minimales *)
  let t = Array.make_matrix n p (-1) in
  (* tableau des provenances *)
  let pr = Array.make_matrix n p (-1, -1) in
  for i = 0 to n-1 do
    for j = 0 to p-1 do
      let res, prec =
        match (i,j) with
        | (0,0) -> m.(0).(0), (-1, -1)
        | (0, _) -> t.(0).(j-1) + m.(0).(j), (0,j-1)
        | (_, 0) -> t.(i-1).(0) + m.(i).(0), (i-1,0)
        | _ ->
          if t.(i-1).(j) < t.(i).(j-1)
          then t.(i-1).(j) + m.(i).(j), (i-1, j)
          else t.(i).(j-1), (i, j-1)
```

```
      in
      t.(i).(j) <- res;
      pr.(i).(j) <- prec
    done
  done;
  let rec chemin i j acc =
    match pr.(i).(j) with
    | (-1, -1) -> acc
    | (k, l) -> chemin k l ((i,j)::acc)
  in
  chemin (n-1) (p-1) []
;;
```

En outre, dans le cas où i et j sont non nuls, il est possible de déterminer la case précédente du chemin à partir de $s_{min}(i-1,j)$ et $s_{min}(i,j-1)$.

On peut donc se contenter de construire t, puis utiliser t pour déterminer un chemin de somme minimale.

```
let smin_it4 m =
  let n = Array.length m in
  let p= Array.length m.(0) in
  (* tableau des sommes minimales *)
  let t = Array.make_matrix n p (-1) in
  for i = 0 to n-1 do
    for j = 0 to p-1 do
      let res =
        match (i,j) with
        | (0,0) -> m.(0).(0)
        | (0, _) -> t.(0).(j-1) + m.(0).(j)
        | (_, 0) -> t.(i-1).(0) + m.(i).(0)
        | _ ->
           if t.(i-1).(j) < t.(i).(j-1)
           then t.(i-1).(j) + m.(i).(j)
           else t.(i).(j-1)
      in
      t.(i).(j) <- res
    done
  done;
```

```
  let rec chemin i j ch =
    let new_ch = (i,j)::ch in
    match (i, j)  with
    | (0, 0)-> new_ch
    | (0, _) -> chemin 0 (j-1) new_ch
    | (_, 0) -> chemin (i-1) 0  new_ch
    | _ ->
       if t.(i-1).(j) + m.(i).(j) = t.(i).(j)
       then chemin (i-1) j new_ch
       else chemin i (j-1) new_ch
  in
  chemin (n-1) (p-1) []
;;
```

2 Principes de la programmation dynamique

La programmation dynamique, comme la méthode « diviser pour régner », résout des problèmes en combinant des solutions de sous-problèmes.

Néanmoins, les sous-problèmes ne sont plus traités de manière indépendante : on mémorise les solutions des problèmes déjà résolus pour éviter d'avoir à les calculer plusieurs fois.

La programmation dynamique est surtout utilisée pour des problèmes d'optimisation : étant donnée une fonction $f : \mathcal{U} \to \mathbb{Z}$ où l'univers $\mathcal{U}$ est fini mais grand, on souhaite déterminer $\max_{y\in\mathcal{U}} f(y)$ ou encore $x \in \mathcal{U}$ tel que $f(x) = \max_{y\in\mathcal{U}} f(y)$.

On procède généralement en quatre étapes :

- Identifier une sous-structure optimale ; il s'agit de voir si la connaissance de $x \in \mathcal{U}$ tel que $f(x) = \max_{y\in\mathcal{U}} f(y)$ permet de déduire des solutions à des sous-problèmes consistant à trouver $x_i \in \mathcal{U}_i$ tels que $f(x_i) = \max_{y\in\mathcal{U}_i} f_i(y)$. Les problèmes associés aux $(f_i,\mathcal{U}_i)$ doivent être plus simples à résoudre que le problème initial.
- Écrire la valeur optimale en fonction de valeurs optimales de sous-problèmes.
- Calculer la valeur d'une solution optimale en stockant les valeurs optimales de sous-problèmes, de manière ascendante (ou *bottom-up*).
- Éventuellement, ajouter des informations pour trouver une solution optimale ou la reconstruire.

3 Exercices

Exercice 4 (Problème du sac-à-dos) *Ce problème se pose de la manière suivantes : étant donnés n objets de valeurs entières $v_1, \dots, v_n$ et de poids respectifs entiers $w_1, \dots, w_n$, comment remplir un sac-à-dos en maximisant la valeur emportée si ce dernier peut contenir au plus un poids W ?*

Autrement dit, on cherche à maximiser $\sum_{i \in I} v_i$ avec la contrainte $\sum_{i \in I} w_i \leqslant W$, pour I une partie de $[\![1, n]\!]$.

On introduit pour deux paramètres $w \in [\![0, W]\!]$ et $j \in [\![0, n]\!]$ le sous-problème dans lequel le sac-à-dos a une capacité de w et les objets disponibles sur les objets 1 à j. On note $K(w, j)$ la solution de ce sous-problème : c'est donc la valeur maximale qu'un sac-à-dos de capacité w peut contenir en choisissant parmi les objets 1 à j ?

1. *Donner pour tout $w \in [\![0, W]\!]$ la valeur de $K(w, 0)$.*
2. *Pour $j \in [\![1, n]\!]$ et $w \in [\![0, W]\!]$, exprimer $K(w, j)$ en fonction de valeurs $K(w', j-1)$ avec $w' \leqslant w$.*
3. *En déduire un algorithme de programmation dynamique calculant la valeur de $K(W, n)$. Quelle est sa complexité ?*
4. *Modifier cet algorithme pour qu'il fournisse la liste des objets à choisir.*
5. *Donner un algorithme calculant la valeur de $K(W, n)$ n'utilisant qu'une place mémoire en $O(W)$. Peut-on trouver la liste des objets à emporter ?*

Exercice 5 (Distance d'édition) *On considère deux mots x et y fixés, de longueurs respectives m et n, sur un même alphabet fini Σ.*

On cherche à aligner *ces deux mots, c'est-à-dire à trouver la plus petite suite d'opérations qui permettent de transformer un mot en l'autre. Les opérations disponibles sont l'insertion d'une lettre , la suppression d'une lettre ou la substitution d'une lettre par une autre.*

Par exemple, voici deux manières d'aligner les mots EXPONENTIEL et POLYNOMIAL

$$\begin{array}{cccccccccccc} E & X & P & O & N & E & N & - & T & I & E & L \\ - & - & P & O & L & Y & N & O & M & I & A & L \end{array}$$

$$\begin{array}{cccccccccccc} E & X & P & O & N & E & N & T & I & E & - & L \\ P & O & L & Y & N & O & M & - & I & - & A & L \end{array}$$

Les tirets représentent l'insertion ou la suppression d'un caractère. On considère que chaque opération coûte 1. On considère que lorsque deux lettres sont identiques, le coût de la vérification est nul. Ainsi, le coût des deux transformations précédentes sont respectivement 7 et 9.

On appelle distance d'édition *entre les deux mots x et y le coût minimal d'une suite d'opérations transformant x en y.*

On pourra vérifier que la complexité dans le pire des cas d'un algorithme naïf est exponentielle.

On considère donc le sous-problème consistant à trouver la distance d'édition $E(i, j)$ entre un préfixe $x[1..i]$ de x est un préfixe $y[1..j]$ de y.

1. *Construire un algorithme de programmation dynamique permettant de calculer la distance d'édition entre x et y.*
2. *Modifier cet algorithme pour afficher la suite des opérations à appliquer à x pour obtenir y.*
3. *Finalement, on veut économiser de la place mémoire. Trouver une manière de calculer $E(m, n)$ en utilisant une place mémoire en $O(\min(m, n))$. Peut-on ensuite trouver la suite d'opérations à appliquer ?*