PROGRAMMATION DYNAMIQUE 2

Des problèmes concrets résolus par programmation dynamique.

1 Problème du sac à dos

Mon sac a une capacité limitée notée M (comme masse). Les n objets, numérotés de 1 à n, que je peux transporter ont une masse $m_i > 0$, mais aussi une valeur $v_i > 0$.

Mon objectif est de transporter la plus grande valeur possible dans mon sac. En clair, je cherche à maximiser la somme $S(I) = \sum_{i \in I} v_i$ (où I est un sous-ensemble de $\{1, \ldots, n\}$) sans dépasser la capacité de mon sac, c'est-à-dire avec la condition $L(I) = \sum_{i \in I} m_i \leqslant M$. On appelle ce genre de problème « problème d'optimisation sous contrainte ».

On note $KP(i, M)$ le problème du problème du sac à dos qu'on remplit avec au plus les i premiers objets et la masse limite M.

Une solution du problème $KP(i, M)$ est un sous-ensemble I de $\{1, \ldots, i\}$ tel que

$$S(I) = \max\{S(J) \ / \ J \subset \{1, \ldots, i\} \text{ et } L(I) \leqslant M\}$$

Si $i = 0$, alors le problème $KP(0, M)$ a pour solution l'ensemble vide de somme nulle.

Si $i \geqslant 1$, deux cas se présentent :

- la solution ne contient pas l'objet numéro i, donc l'ensemble I est solution du problème $KP(i-1, M)$;
- la solution contient l'objet numéro i, donc les autres objets de la solution (*i.e.* l'ensemble $I - \{i\}$) forment une solution du problème $KP(i-1, M-m_i)$ (réfléchir quelques instants pour s'en convaincre par l'absurde) : si on est dans ce cas, alors $M \leqslant m_i$.

Le principe de sous-optimalité de Bellmann est satisfait.

Donc si on note I' la solution du problème $KP(i-1, M)$ et si $M \geqslant m_i$, I'' la solution du problème $KP(i-1, M-m_i)$, alors on pose I selon les cas : si $S(I') > S(I'') + v_i$, alors on pose $I = I'$, sinon $I = I'' \cup \{i\}$

et donc $S(I) = \max\{S(I'), S(I'') + v_i\}$.

Dans le cas où $M < m_i$, on ne peut bien sûr calculer que I'.

On a ainsi une formulation récursive de l'algorithme de résolution du problème KP. Comme on risque de recalculer plusieurs fois les mêmes sous-problèmes, on peut tenter une résolution par programmation dynamique.

On représente les masses m_i et les valeurs v_i par des tableaux de nombres (disons des entiers pour fixer les idées) (i commence à 0, on décale les indices...). Une solution sera représentée par une liste d'indices.

D'abord on cherche la valeur maximale de la somme.

```
let kp m v maxi =
  let n = vect_length m in
  let sommes = make_matrix (n) (maxi +1) (-1) in
  let rec kpaux i maxi =
    if i <= -1 then 0
    else begin
      let s = sommes.(i).(maxi) in
      if s != -1 then s
      else begin
        let s' = kpaux (i-1) maxi in
        let s =
            if m.(i) <= maxi then
              let s'' = kpaux (i-1) (maxi - m.(i)) in
              max s' (s'' + v.(i))
            else
```

```
                s'
        in
        sommes.(i).(maxi) <- s; s
      end
    end
  in
  kpaux (n-1) maxi;;
```

Pour obtenir la solution, il suffit d'ajouter une information supplémentaire dans la table de mémoïsation.

```
let kp m v maxi =
  let n = vect_length m in
  let sommes = make_matrix (n) (maxi +1) (-1, []) in
  let rec kpaux i maxi =
    if i <= -1 then 0,[]
    else begin
      let s, l = sommes.(i).(maxi) in
      if s != -1 then (s, l)
      else begin
        let s', l' = kpaux (i-1) maxi in
        let s, l =
            if m.(i) <= maxi then
              let s'', l'' = kpaux (i-1) (maxi - m.(i)) in
              if s' > s'' + v.(i) then (s', l') else (s'' + v.(i), i :: l'')
            else
              (s', l')
        in
        sommes.(i).(maxi) <- (s, l); (s, l)
      end
    end
  in
  kpaux (n-1) maxi;;
```

La complexité est en $O(nM)$.

2 Ordonnancement de tâches pondérées

On considère un ensemble de tâches à accomplir, qui débutent à l'instant d_i et finissent à l'instant f_i et qui ont chacune une valeur v_i.

Deux tâches sont compatibles si elles sont disjointes. Un ensemble de tâches est dit compatible si les tâches de cet ensemble sont deux à deux compatibles. Le but est de trouver un ensemble de tâches compatible I dont la valeur totale $S(I) = \sum_{i \in I} v_i$ est maximale.

D'abord on ordonne les tâches par heures de fin croissantes : $f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n$. Puis on calcule une famille d'indices p_i : p_i est l'indice maximal $j < i$ tel que la tâche j soit compatible avec la tâche i.

On note $OTP(i)$ le problème d'ordonnancement des tâches 1 à i.

Une solution du problème $OTP(i)$ est un sous-ensemble I de $\{1, \ldots, i\}$ tel que

$$S(I) = \max\{S(J) \ / \ J \subset \{1, \ldots, i\} \text{ et } I \text{ compatible}\}$$

Si $i = 0$, le problème $OTP(0)$ a pour solution l'ensemble vide.

Si $i \geqslant 1$, deux cas se présentent :

- la solution ne contient pas la tâche numéro i, donc l'ensemble I est solution du problème $OTP(i-1)$;
- la solution contient l'objet numéro i, donc les autres tâches de la solution (*i.e.* l'ensemble $I \subset \{i\}$) forment une solution du problème $OTP(p_i)$, car elles doivent être compatibles avec la tâche numéro i et de numéro $j < i$.

Ici encore, le principe de sous-optimalité de Bellmann est satisfait.

Donc si on note I' la solution du problème $OTP(i-1)$ et I'' la solution du problème $OTP(p_i)$, alors on pose I selon les cas : si $S(I') > S(I'') + v_i$, alors on pose $I = I'$, sinon $I = I'' \cup \{i\}$

et donc $S(I) = \max\{S(I'), S(I'') + v_i\}$.

On a ainsi une formulation récursive de l'algorithme de résolution du problème OTP. Comme on risque de recalculer plusieurs fois les mêmes sous-problèmes, on peut tenter une résolution par programmation dynamique.

On représente les tâches par un tableau t de couples (d_i, f_i) et les valeurs v_i par un tableau de nombres (i commence à 0, on décale les indices...). Une solution sera représentée par une liste d'indices.

On cherche la valeur maximale de la somme et l'ensemble associé.

```
let otp t v =
  let n = vect_length t in
  let p = make_vect n (-1) in
  let sommes = make_vect n (-1,[]) in
  for i = 1 to (n-1) do
    let q = ref 0 in
    while snd t.(!q) <=  fst t.(i) do
      incr q
    done;
    p.(i) <- !q - 1
  done;
  let rec otpaux i =
    if i = -1 then (0,[])
    else
      let s, l = sommes.(i) in
      if s != -1 then (s, l)
      else begin
        let s, l =
          let s', l' = otpaux (i-1) and s'', l'' = otpaux p.(i) in
          if s' > s'' + v.(i) then (s', l')
          else (s''+ v.(i), i :: l'')
        in
        sommes.(i) <- (s, l);
        (s, l)
      end
  in
  otpaux (n-1);;
```

3 Distance d'édition et alignement de séquences

Pour transformer un mot en un autre, on s'autorise les transformations suivantes :

- on change une lettre en une autre,
- on supprime une lettre,
- on insère une lettre.

On remarque que ces transformations sont réversibles, donc si on peut passer du mot A au mot B, par le même nombre de transformations, on peut passer de B à A.

On appelle distance d'édition entre deux mots le nombre minimal de transformations nécessaires.

Par exemple, pour passer de `"chien"` à `"miels"`, on peut faire les transformations :

`"chien"` → `"hien"` → `"mien"` → `"miel"` → `"miels"`

En fait, pour mieux les visualiser, on ajoute le symbole `"-"` parmi les lettres :

```
chien-
-hien-
-mien-
-miel-
-miels
```

Pour toute suite de transformations, on peut trouver un alignement des mots qui permet de visualiser les transformations.

L'objectif est de donner un algorithme qui calcule la distance d'édition et qui donne l'alignement correspondant.

Soit A, B deux mots de longueur $n \geqslant 1$, $p \geqslant 1$ respectivement. On note $d(A, B)$ la distance d'édition entre A et B.

Pour passer de A à B, il y a plusieurs possibilités :

a) soit les deux mots ont été alignés sur leurs premières lettres ; cela peut signifier deux choses : si les premières lettres sont égales, la première de A n'a pas été modifiée, sinon elle a été changé en la première de B

b) soit la première lettre du mot A a été aligné avec un -

c) soit la première lettre du mot B a été aligné avec un -

On note A' et B' les suffixes de longueur $n-1$, $p-1$ de A, B respectivement, et a, b les premières lettres de A, B respectivement. Alors selon l'alignement des mots A et B, on a les relations suivantes :

a) $d(A, B) = 1 + d(A', B')$ si $a \neq b$, $d(A, B) = d(A', B')$ si $a = b$

b) $d(A, B) = 1 + d(A', B)$

c) $d(A, B) = 1 + d(A, B')$

On note $\delta(a, b) = 0$ si $a = b$, $\delta(a, b) = 1$ si $a \neq b$.

Alors $d(A, B) = \min(d(A', B') + \delta(a, b), d(A', b), d(A, B'))$.

Il reste les cas de bases : si A est le mot vide, $d(A, B) = long(B)$ et si B est le mot vide, $d(A, B) = long(A)$.

Remarque : le principe de sous-optimalité de Bellmann s'applique ici, car si on a un alignement optimal de A et B, alors l'alignement induit sur (A', B') ou (A', B) ou (A, B') est aussi optimal.

On représente les mots par des chaînes de caractères, et plutôt que de travailler sur les chaînes elles-mêmes, on travaille sur les « tranches de chaînes ».

```
let rec mini l =
  match l with
  | [a] -> a
  | a :: q -> min a (mini q);;

let distance l m =
 let n = string_length l and p = string_length m in
 let t = make_matrix n p (-1) in
 let rec dist i j =
   if i = n then p - j
   else if j = p then n - i
   else
     let d = t.(i).(j) in
     if d != -1 then d
     else begin
       let delta = (if l.[i] = m.[j] then 0 else 1) in
       let ds = [delta + dist (i+1) (j+1); 1 + dist (i+1) j; 1 + dist i (j+1)] in
       let d = mini ds in
       t.(i).(j) <- d;
       d
     end
 in
 dist 0 0;;
```

Pour obtenir en plus l'alignement associé, il faudrait qu'on ajoute une donnée dans les mémoires (une liste) : dans les cas de bases, on doit alors transformer une chaîne en une liste de caractères, ce qui peut être fait par une fonction auxiliaire. Cette fonction a une complexité linéaire et doit être appliquée pour chaque cas de base : il y en a $n+p-1$.

```
let transforme ch i j =
  let rec traux k l =
    if k = j then l
    else traux (k+1) (ch.[k] :: l)
  in
  rev (traux i []);;

let rec mini l =
  match l with
  | [x] -> x
  | (a, b, b') :: q -> let (c, d,d') = mini q in
    if a < c then (a, b,b')
    else (c, d,d');;

let distance l m =
 let n = string_length l and p = string_length m in
 let t = make_matrix (n+1) (p+1) (-1, [], [])  in (*1*)
  let rec dist i j =
   if i = n then (p - j, [], transforme m j p)
   else if j = p then (n - i, transforme l i n, [])
   else let (d, al, bl) = t.(i).(j) in
     if d != -1 then (d, al, bl)
     else let (d, al, bl) =
          if i = n then let (d', al', bl') = dist n (j+1)
            in (1 + d', '-' :: al, m.[j] :: bl)
          else if j = p then let (d', al', bl') = dist (i+1) p
            in (1 + d', l.[i] :: al, '-' :: bl)
          else (
            let delta = (if l.[i] = m.[j] then 0 else 1) in
            let (d', al', bl') = dist (i + 1) (j + 1) in
            let (d'', al'', bl'') = dist (i + 1) j in
            let (d''', al''', bl''') = dist i (j + 1) in
            let ds = [(delta + d', l.[i] :: al', m.[j] :: bl');
                  (1 + d'', l.[i] :: al'' , '-' :: bl'' );
                  (1 + d''', '-' :: al''' , m.[j] :: bl''' )] in
            mini ds
          )
          in (*3*)
     t.(i).(j) <- (d, al, bl);
     (d, al, bl)
 in (*2*)
 dist 0 0;;
```

Pour éviter cette fonction auxiliaire et diminuer la complexité globale de l'algorithme, on modifie les cas de base pour que cette transformation se fasse au fur et à mesure et en ajoutant un caractère spécial en fin de chaîne qui est supposé ne jamais apparaître dans les mots pour forcer le calcul de l'alignement jusqu'au bout.

```
let distance l m =
 let l = l ^ "@" and m = m ^ "@" in
 let n = string_length l and p = string_length m in
 let t = make_matrix (n+1) (p+1) (-1, [], [])  in
  let rec dist i j =
```

```
  if i = n && j = p then (0, [], [])
  else let (d, al, bl) = t.(i).(j) in
    if d != -1 then (d, al, bl)
    else let (d, al, bl) =
        if i = n then let (d', al', bl') = dist n (j+1)
          in (1 + d', '-' :: al, m.[j] :: bl)
        else if j = p then let (d', al', bl') = dist (i+1) p
          in (1 + d', l.[i] :: al, '-' :: bl)
        else (
          let delta = (if l.[i] = m.[j] then 0 else 1) in
          let (d', al', bl') = dist (i + 1) (j + 1) in
          let (d'', al'', bl'') = dist (i + 1) j in
          let (d''', al''', bl''') = dist i (j + 1) in
          let ds = [(delta + d', l.[i] :: al', m.[j] :: bl');
                (1 + d'', l.[i] :: al'' , '-' :: bl'' );
                (1 + d''', '-' :: al''' , m.[j] :: bl''' )] in
          mini ds
        )
        in
    t.(i).(j) <- (d, al, bl);
    (d, al, bl)
in
dist 0 0;;
```