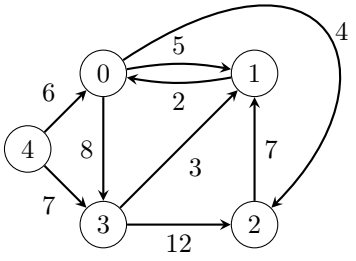
Chapitre 6 : Introduction à la programmation dynamique

1 Introduction

La programmation dynamique [1] 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 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 bien. Citons quelques exemples concrets :

— Dans le graphe suivant, quel est le poids d'un plus court chemin [2] entre le sommet 3 et le sommet 2 ? Cette question est au programme de deuxième année.



— Dans la matrice suivante, on s'intéresse au chemin de la case en haut à gauche à celle en bas à droite, utilisant seulement les déplacements $\to$ et $\downarrow$, qui maximise la somme des entiers rencontrés sur le chemin. Quel est ce chemin et son poids ?

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

Ce problème sera résolu dans ce chapitre.

— On appelle sous-séquence commune à deux chaînes de caractères s et t une chaîne x dont les caractères apparaîssent dans le même ordre dans s et dans t (avec possiblement des caractères intercalés). Quelle est la (une) plus longue sous-séquence commune à « arythmie » et « rhomboédrique » ?

— etc...

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 technique la plus efficace.

2 Un exemple complet : chemin de poids maximal dans une matrice

2.1 Le problème

On reprend le problème cité plus haut, 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 déplacements $\to$ et $\downarrow$, dont le *poids* (somme des entiers rencontrés) est maximal.

1. qui n'est pas une méthode de programmation...
2. Naturellement le poids d'un chemin est la somme des poids des arcs qui composent ce chemin.

2.2 Recherche exhaustive ?

On peut éventuellement chercher à examiner tous les chemins possibles, car ils sont en nombre fini. Dénombrons les : 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. Donnons un équivalent asymptotique de cette quantité lorsque $n = m$, grâce à la formule de Stirling :

$$N_{n,n} = \binom{2n-2}{n-1} = \frac{(2n-2)!}{(n-1)!^2} \underset{n\to+\infty}{\sim} \frac{(2n-2)^{2n-2}\,\mathrm{e}^{-2(n-1)}\sqrt{2\pi(2n-2)}}{2(n-1)^{2(n-1)}\,\mathrm{e}^{-2(n-1)}\,\pi(n-1)} = \frac{2^{2n-2}}{\sqrt{\pi(n-1)}} \underset{n\to+\infty}{\sim} \frac{2^{2n-2}}{\sqrt{\pi n}}$$

Le nombre de chemins possibles est donc exponentiel en n, déja pour $n = 30$ un algorithme qui explore tous les chemins possibles est impraticable.

2.3 Solutions aux sous-problèmes

Considérons une solution à notre problème, c'est-à-dire un 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 $(0,0) \overset{c_1}{\rightsquigarrow} (i,j) \overset{c_2}{\rightsquigarrow} (n-1,m-1)$. Les deux chemins c_1 et c_2 sont des chemins de poids maximaux de la case $(0,0)$ à (i,j) et de la case (i,j) à $(n-1,m-1)$.

La technique de démonstration est classique, et à retenir : supposons que c_1 ne soit pas optimal, il existe donc un chemin c_1' de poids strictement supérieur de la case $(0,0)$ à la case (i,j). Alors le chemin $(0,0) \overset{c_1'}{\rightsquigarrow} (i,j) \overset{c_2}{\rightsquigarrow} (n-1,m-1)$ est un chemin de poids strictement supérieur à c, ce qui est absurde. De même pour c_2.

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 (en effet comme on se restreint aux déplacements $\downarrow$ et $\rightarrow$, tout se passe sur une matrice de taille $(i+1)\times(j+1)$).

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

2.4 Une relation récursive pour le poids maximal d'un chemin

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 $(p_{i,j})_{0\leq i<n,0\leq j<m}$ satisfont la relation suivante :

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

En effet :

- la relation pour les trois premiers points est évidente, car dans ce cas il n'y a qu'un seul chemin licite de la case $(0,0)$ à la case (i,j). On suppose dorénavant $i>0$ et $j>0$;
- À partir d'un chemin $\mathcal{C}$ menant à la case $(i-1,j)$, on en construit un menant à la case (i,j) via un déplacement $\downarrow$. En choisissant $\mathcal{C}$ de poids maximal $p_{i-1,j}$, on obtient $p_{i,j} \geq a_{i,j} + p_{i-1,j}$. Comme on obtient de manière symétrique $p_{i,j} \geq a_{i,j} + p_{i,j-1}$, on conclut que $p_{i,j} \geq a_{i,j} + \max\{p_{i-1,j}, p_{i,j-1}\}$;
- Réciproquement, tout chemin licite 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 la case $(i-1,j)$ ou à la case $(i,j-1)$. Ainsi $\max\{p_{i-1,j}, p_{i,j-1}\} \geq p_{i,j} - a_{i,j}$.

Et la relation est démontrée. 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.

2.5 Un calcul itératif des $p_{i,j}$

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 $O(nm)$ à l'aide des coefficients de la matrice A en suivant la relation précédente. Voici un code Caml :

```
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
;;
```

L'appliquer à la matrice A de l'introduction donne la matrice P suivante :

$$A = \begin{pmatrix} 2 & 39 & 12 & 49 & 47 & 18 & 22 & 19 \\ 37 & 21 & 34 & 26 & 10 & 2 & 35 & 39 \\ 31 & 21 & 12 & 26 & 34 & 27 & 7 & 22 \\ 20 & 46 & 16 & 2 & 11 & 40 & 36 & 13 \\ 18 & 30 & 32 & 37 & 28 & 24 & 9 & 6 \end{pmatrix} \quad \text{et} \quad P = \begin{pmatrix} 2 & 41 & 53 & 102 & 149 & 167 & 189 & 208 \\ 39 & 62 & 96 & 128 & 159 & 169 & 224 & 263 \\ 70 & 91 & 108 & 154 & 193 & 220 & 231 & 285 \\ 90 & 137 & 153 & 156 & 204 & 260 & 296 & 309 \\ 108 & 167 & 199 & 236 & 264 & 288 & 305 & 315 \end{pmatrix}$$

2.6 Détermination d'une solution au problème initial

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, de la case $(0,0)$ à la case $(n-1,m-1)$.

Parmi plusieurs solutions, on propose la suivante : il suffit de remonter de la case $(n-1,m-1)$ à la case $(0,0)$ dans la matrice P. Depuis une case (i,j), on a le choix entre remonter à la case $(i-1,j)$ et remonter 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 donc un chemin convenable avec une complexité supplémentaire $O(n+m)$.

Terminons cet exemple par une implémentation. On encode un chemin comme une liste de caractères « `>` » ou « `v` » indiquant à chaque étape s'il faut se diriger sur la case de droite ou celle d'en dessous. Comme on remonte depuis la case $(n-1,m-1)$ insérer successivement les caractères dans une liste fait l'affaire. Voici le code :

```
let max_chemin a=
  let n,m=Array.length a, Array.length a.(0) in
  let p=calcul_p a in
  let i=ref (n-1) and j=ref (m-1) and q=ref [] in
  while !i>0 && !j>0 do
    if p.( !i-1).( !j) > p.( !i).( !j-1) then
      begin q:= "v":: !q ; decr i end
    else
      begin q:= ">":: !q ; decr j end
  done ;
  while !i>0 do
    q:="v":: !q ; decr i
  done ;
  while !j>0 do
    q:=">":: !q ; decr j
  done ;
  !q
;;
```

Remarquez que les deux dernières boucles `while` servent simplement à remonter d'un des bords (gauche ou supérieur) à la case initiale, et seule l'une des deux est utile. Appliquons l'algorithme à la matrice A de l'exemple :

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

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

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

Le lecteur non convaincu vérifiera que ce chemin est de poids 315, ce qui correspond à $p_{4,7} = p_{n-1,m-1}$.

3 Principes de la programmation dynamique, et variantes

L'exemple précédent est typique d'une résolution de problème par programmation dynamique. Donnons un résumé de la démarche, dans un cadre plus abstrait.

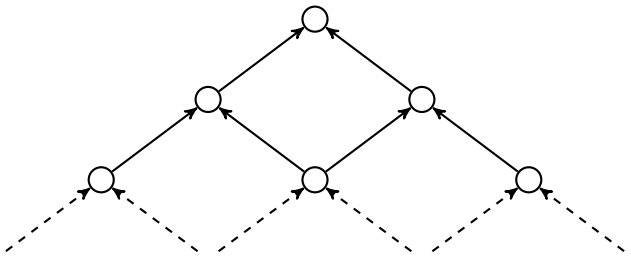
3.1 La démarche d'une résolution de problème par programmation dynamique

On se donne donc $f : \mathcal{U} \to \mathbb{Z}$, 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 $M_{f,\mathcal{U}}$ la quantité $\max_{y\in\mathcal{U}}\{f(y)\}$.

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

1. Identifier une *sous-structure optimale* : c'est un indice de l'application de la programmation dynamique. Il s'agit de voir que si l'on 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 $f_i(x_i) = \max_{y\in\mathcal{U}_i}\{f(y)\} = M_{f_i,\mathcal{U}_i}$ ». Les problèmes associés aux $(f_i,\mathcal{U}_i)$ doivent être plus simples à résoudre que le problème associé à $(f,\mathcal{U})$. Dans l'exemple précédent, les problèmes $(f_i,\mathcal{U}_i)$ étaient des déterminations de chemins de poids maximal dans des matrices plus petites.
2. Déduire de la sous-structure optimale une relation récursive permettant le calcul de $M_{f,\mathcal{U}}$ à partir de certains $M_{f_i,\mathcal{U}_i}$. Dans l'exemple précédent, on a exhibé une relation entre $p_{n-1,m-1}$, $p_{n-2,m-1}$ et $p_{n-1,m-2}$.

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 tel quel $M_{f,\mathcal{U}}$ mène en général à une solution très coûteuse, ce qui peut être résumé au travers du schéma suivant, calqué sur le problème étudié précédemment :



3. Pour pallier le problème évoqué au point (2), on calcule alors les $M_{f_i,\mathcal{U}_i}$ utiles (y compris $M_{f,\mathcal{U}}$) itérativement, en faisant usage d'un tableau pour stocker tous ces éléments. On peut parfois se contenter de n'en stocker que certains, par exemple dans le problème précédent un espace $O(n)$ en plus de la matrice A (au lieu de $O(nm)$) suffirait [3] pour calculer $p_{n-1,m-1}$.
4. Enfin, on modifie légèrement le calcul des $M_{f_i,\mathcal{U}_i}$ pour obtenir en même temps [4] pour tout i un x_i satisfaisant $f_i(x_i) = M_{f_i,\mathcal{U}_i}$. En général cette étape n'est pas difficile.

3.2 Une parenthèse sur les problèmes de combinatoire

La technique évoquée ci-dessus 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 $\mathcal{U}$. La démarche ressemble à celle ci-dessus :

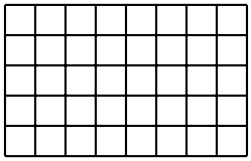
3. Comment procéder ?
4. Dans le problème précédent, on a choisi de ne calculer un chemin convenable qu'après le calcul des $p_{i,j}$, mais on aurait pu par exemple stocker en parallèle des $p_{i,j}$ (dans une troisième matrice) le dernier déplacement à effectuer pour arriver en (i,j) en suivant un chemin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

1. On partitionne l'ensemble $\mathcal{U}$ 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 $\mathcal{U} = \cup_i \widetilde{\mathcal{U}_i}$ (l'union étant disjointe) 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.

Pour résumer, la résolution d'un problème de combinatoire suit essentiellement les points (1) à (3) évoqués pour la résolution d'un problème d'optimisation par programmation dynamique, le point (4) n'ayant pas de sens ici. Détaillons rapidement deux exemples.

3.2.1 Nombre de chemins sur un quadrillage

On considère un quadrillage de taille $n \times m$, comme celui-ci dessous :



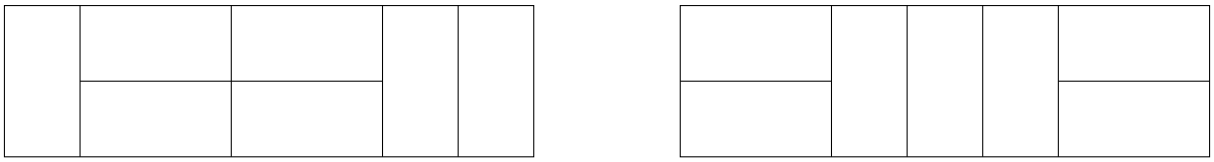
On cherche le nombre de chemins sur partant du coin en haut à gauche jusqu'au coin en bas à droite, en suivant seulement les directions [5] $\rightarrow$ et $\downarrow$. Résolvons le rapidement :

1. On peut indexer les points de la grille de $(0,0)$ (en haut à gauche) à (n,m) (en bas à droite). Notons $\mathcal{C}_{i,j}$ l'ensemble des chemins de $(0,0)$ à (i,j), utilisant seulement les déplacements autorisés. Alors pour tout $i,j \geq 0$, on a
$$\mathcal{C}_{i,j} = \widetilde{\mathcal{C}_{i-1,j}} \cup \widetilde{\mathcal{C}_{i,j-1}}$$
où $\widetilde{\mathcal{C}_{i-1,j}}$ est l'ensemble des chemins de $\mathcal{C}_{i-1,j}$, complétés par le segment $(i-1,j) \rightarrow (i,j)$, et de même pour $\widetilde{\mathcal{C}_{i,j-1}}$. On convient que $\mathcal{C}_{-1,j} = \mathcal{C}_{i,-1} = \emptyset$ et que $C_{0,0}$ contient comme unique élément le chemin réduit au point $(0,0)$.
2. En notant $N_{i,j} = |\mathcal{C}_{i,j}|$, on a donc la relation de récurrence $N_{i,j} = N_{i-1,j} + N_{i,j-1}$, valable pour $i,j > 0$, sinon $N_{i,0} = N_{0,j} = 1$.
3. On peut donc tabuler les $N_{i,j}$ dans un tableau de taille $(n+1) \times (m+1)$, car c'est $N_{n,m}$ qui nous intéresse.

Le lecteur pourra vérifier qu'il y a 1287 chemins convenables, pour l'exemple [6] de la grille de taille 5×8.

3.2.2 Un problème de pavage

On considère un rectangle de taille $2 \times n$, et on s'intéresse aux *pavages* de ce rectangle par des dominos 1×2. La figure suivante montre deux exemples de pavage d'un rectangle 2×7.



Notons F_n le nombre de pavages possibles d'un rectangle de taille $2 \times n$. Cherchons une relation de récurrence nous permettant de calculer F_n efficacement. Supposons $n \geq 2$ et considèrons le domino occupant le coin en haut à droite du rectangle.

— si ce domino est placé verticalement (comme dans le pavage de gauche dans la figure ci-dessus), alors il reste à paver un rectangle $2 \times (n-1)$: le nombre de tels pavages est donc F_{n-1} ;

— sinon, le domino est placé horizontalaement (comme dans le pavage de droite). Nécessairement, un autre domino horizonal est placé en dessous, il reste donc à paver un rectangle de taille $2 \times (n-2)$, et il y a F_{n-2} tels pavages.

5. Oui, ce problème de combinatoire ressemble fortement au problème d'optimisation vu précédemment : c'est fait exprès !
6. Comme déja évoqué, il y a en fait $\binom{n+m}{n}$ chemins possibles, et $1287 = \binom{13}{5}$...

Ainsi, $(F_n)_n$ satisfait la relation de récurrence $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$ (on reconnaît la suite de Fibonacci...), avec conditions initiales $F_0 = F_1 = 1$. Comme on l'a vu au chapitre 3, il vaut mieux faire usage d'un tableau que de récursivité pour calculer efficacement F_n.

Remarque 1. *Ce problème était facile. Le lecteur pourra chercher le nombre de pavages possibles d'un rectangle* 3×30 *avec des dominos* 1×2*, c'est moins évident !*

Avant de voir d'autres exemples de résolution de problèmes d'optimisation à l'aide de la programmation dynamique, parlons brièvement des méthodes gloutonnes.

3.3 Algorithmes « glouton »

3.3.1 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 la relation satisfaite par $p_{i,j}$, poids d'un chemin maximal de $(0,0)$ à (i,j), était $p_{i,j} = a_{i,j} + \max\{p_{i-1,j}, p_{i,j-1}\}$. Faisons un choix (localement optimal), et décidons de choisir $(i',j') \in \{(i-1,j),(i,j-1)\}$ tel que $a_{i',j'}$ est maximal. En faisant systématiquement ce choix (le choix glouton) à chaque étape depuis la case $(n-1,m-1)$ jusqu'à la case $(0,0)$, on construit directement un unique chemin. La figure suivante rappelle le chemin de poids maximal (315) trouvé grâce à la programmation dynamique, et le chemin fourni par le choix glouton.

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

On vérifie que le chemin donné par l'algorithme glouton a un poids de 304, ce qui n'est pas optimal (mais pas loin !).

3.3.2 Principe des algorithmes glouton

Un algorithme glouton pour résoudre un problème d'optimisation suit les mêmes principes que la résolution par programmation dynamique, néanmoins le point (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). Visuellement, on peut représenter les choix faits par un algorithme glouton par le schéma ci-dessous, à droite.

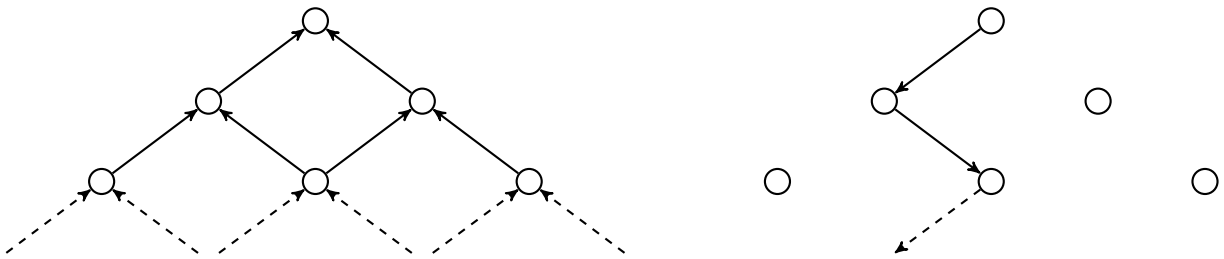


FIGURE 1: Comparaison des stratégies dynamique et gloutonne

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. On verra notamment en deuxième année 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 [7] 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. Par exemple pour le problème précédent, l'algorithme glouton n'a qu'une complexité $O(n+m)$.

7. C'est l'algorihtme de Dijkstra.

4 Deux autres exemples de résolution par programmation dynamique

4.1 Le problème de la sous séquence commune

Ce problème, déja évoqué dans l'introduction, consiste à trouver un mot x qui est une sous-séquence commune maximale à deux mots s et t. Par exemple, une sous-séquence commune maximale à « arythmie » et « rhomboédrique » est « rhmie », de longueur 5. Ce problème a des applications pratiques, notamment en génétique : la proximité de deux individus peut être évaluée en calculant une sous-séquence commune [8] à deux séquences d'ADN prises sur les individus.

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 $\ell_{i,j}$ 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 $\ell_{n,m}$, et une sous-séquence associée. Supposons que l'on connaisse une sous-séquence commune x de longueur $\ell_{n,m}$, 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 (notés s' et t' dans la suite), et c'est même une plus longue sous-séquence commune à ces deux mots (l'argument est classique : 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').

— 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 au deux), et c'est même une plus longue sous-séquence commune de manière évidente.

Nous avons exhibé une sous-structure optimale !

Une relation de récurrence. La discussion précédente nous fournit une relation de récurrence sur les $\ell_{i,j}$, à savoir :

$$\ell_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0\,; \\ 1 + \ell_{i-1,j-1} & \text{si les préfixes de tailles } i \text{ et } j \text{ de } s \text{ et } t \text{ terminent par la même lettre;} \\ \max\{\ell_{i-1,j}, \ell_{i,j-1}\} & \text{sinon.} \end{cases}$$

Calcul itératif des $\ell_{i,j}$ et construction d'une sous-séquence commune. Pour calculer les $(\ell_{i,j})_{0\leq i\leq n, 0\leq j\leq m}$, on procède itérativement en remplissant un tableau de taille $(n+1)\times(m+1)$. De manière similaire au problème du chemin de poids maximal, il suffit de remonter depuis la case (n,m) jusqu'à la case $(0,0)$ (ou plus simplement à un bord supérieur ou inférieur du tableau) pour construire une sous-séquence commune. Voici une implémentation :

```
let plssc s t=
  let n, m=String.length s, String.length t in
  let long=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
        long.(i).(j) <- 1+long.(i-1).(j-1)
      else
        long.(i).(j) <- max long.(i-1).(j) long.(i).(j-1)
    done
  done ;
  let x=String.make long.(n).(m) 'a' and i=ref n and j=ref m and k=ref (long.(n).(m)-1) in
  while !k>=0 do
    if long.( !i).( !j) = long.( !i-1).( !j) then
      decr i
    else if long.( !i).( !j) = long.( !i).( !j -1) then
      decr j
    else begin
      x.[ !k] <- s.[ !i-1] ;
      decr i ;
      decr j ;
      decr k ;
    end
  done ;
  x
;;
```

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

Une fois les $\ell_{i,j}$ calculés, on connaît la longueur d'une plus longue sous-séquence commune. On crée alors une chaîne à la bonne taille, qu'on va modifier [9]. Pour cela, on remonte depuis la case (n, m). Si $\ell_{i,j} = \ell_{i-1,j}$ ou $\ell_{i,j-1}$, 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 $(i-1, j-1)$. Testons :

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

Complexité. La détermination des $\ell_{i,j}$ se fait en temps $O(nm)$, alors que la construction d'une plus longue sous-séquence commune ne prend qu'un temps $O(n+m)$.

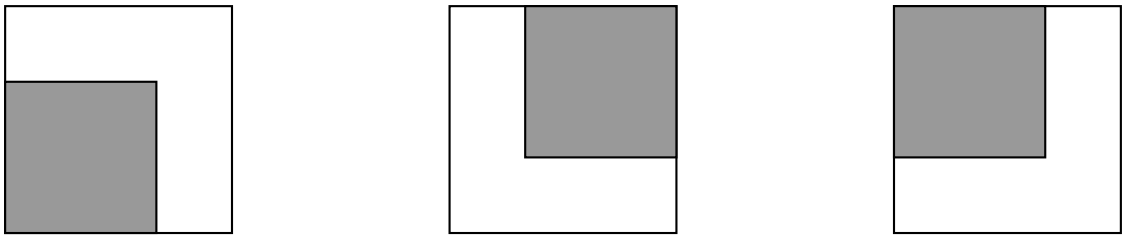
4.2 Plus grand carré de zéros dans une matrice binaire

Pour ce dernier exemple, on se donne une matrice $A = (a_{i,j})_{0 \leq i < n, 0 \leq j < m}$ de taille $n \times m$, constituée de zéros et de uns, comme la suivante :

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

On cherche la taille du plus grand carré de zéros dans cette matrice. Pour l'exemple ci-dessus, la réponse cherchée est trois.

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 $\{(i-1, j), (i, j-1), (i-1, j-1)\}$ sont tous remplis de zéros, comme le montre la figure ci-dessous.



Relation de récurrence. La découverte de la sous-structure optimale nous permet d'exhiber facilement la relation de récurrence suivante, sur la taille du plus grand carré de zéro terminant à l'indice (i, j), qu'on note $t_{i,j}$, en convenant que $t_{-1,j} = t_{i,-1} = 0$:

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

On laisse au lecteur le soin d'implémenter un code permettant de calculer la taille d'un plus grand carré de zéro dans une matrice binaire, et de donner la position de son coin en bas à droite.

9. On rappelle que les chaînes de caractères en Caml sont très semblables à des tableaux de caractères. On accède ou modifie le k-ème caractère de `x` via `x.[k]`, et `String.make` permet, de manière analogue à `Array.make`, de créer une chaîne de caractères de la longueur désirée. Attention, depuis récemment les chaînes de caractères tendent à devenir immuables en Ocaml (le code précédent fonctionne mais avec un avertissement), et un type spécial (Bytes, c'est-à-dire octets) remplace les chaînes mutables. Avec cette version, il faut créer un objet de type Bytes, que l'on convertit à la fin du code en String.