Contrôle Classant – Informatique 431

Claire Kenyon Jean-Jacques Lévy *

Ecole polytechnique, 30 juin 2004

On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction. Lorsqu'une question porte sur une complexité en temps ou en espace, seul l'ordre de grandeur en fonction du paramètre précisé est demandé. En Java, la concaténation `u+v` *de deux chaînes de caractères* `u` *et* `v` *alloue un espace mémoire dont la taille est la somme des longueurs de* `u` *et de* `v`.

Un *polyomino* P est une union de carrés élémentaires du plan, où le carré (i,j) est l'ensemble $C_{i,j}$ de points défini par $C_{i,j} = \{(x,y) \mid i \leq x \leq i+1, j \leq y \leq j+1\}$. Un polyomino est dit *4-connexe* si, pour toute paire de carrés C et C' de P, on peut relier C à C' par une suite de carrés $C = C_0$, C_1, C_2, $\ldots C_\ell = C'$ telle que, pour tout k $(0 \leq k < \ell)$, le carré C_{k+1} soit le voisin Nord, Sud, Est ou Ouest du carré C_k. Un polyomino est *8-connexe* si, pour toute paire de carrés C et C' de P, on peut relier C à C' par une suite de carrés $C = C_0$, C_1, C_2, $\ldots C_\ell = C'$ telle que, pour tout k $(0 \leq k < \ell)$, le carré C_{k+1} soit le voisin Nord, Sud, Est, Ouest, Nord-Ouest, Sud-Ouest, Nord-Est ou Sud-Est du carré C_k. Le complémentaire d'un polyomino P est l'union des carrés élémentaires $C_{i,j}$ non contenus dans P.

Un polyomino est *simplement connexe*, ou encore *sans trou*, s'il est 4-connexe et si son complémentaire dans le plan est 8-connexe. Dans le problème, nous ne considérerons que des polyominos de surface finie non nulle. On identifie les polyominos à translation près ; par exemple il n'existe qu'un seul polyomino simplement connexe de surface 1 et que deux polyominos simplement connexes de surface 2.

Un polyomino P simplement connexe peut être codé par un *mot de contour*, obtenu à partir de tout point de départ à coordonnées entières sur la frontière de P, en parcourant la frontière de P dans le sens direct (sens contraire du mouvement des aiguilles d'une montre) ; pour chaque morceau de contour de longueur 1 parcouru, on écrira g pour gauche, d pour droite, h pour haut et b pour bas. On obtient donc un mot sur l'alphabet $\{g, d, h, b\}$. Ce codage n'est pas unique : il dépend du choix du point de départ.

Un mot de contour est *canonique* si son point de départ est le point de la frontière de P minimum dans l'ordre lexicographique $<_{\ell ex}$ (L'ordre lexicographique est défini par $(x,y) <_{\ell ex} (x',y')$ si et seulement si $x < x'$ ou $x = x'$ et $y < y'$).

Si on pose $a^n = aa\ldots a$ pour le mot formé de n lettres a $(n \geq 0)$, deux contours possibles pour le polyomino de la figure 1(a), à partir des deux points marqués sur la frontière, sont $d^4hdhg^3bghg^3bg^2hg^3bdbd^5hdb$ et $dbd^5hdbd^4hdhg^3bghg^3bg^2hg^3b$. Le deuxième est canonique.

* avec les participations de Luc Maranget, Didier Rémy et Abhishek Thakur

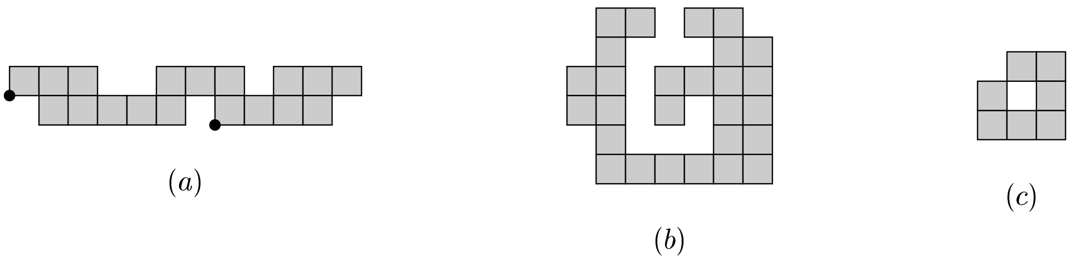


FIG. 1 – Trois polyominos simplement connexes

Partie I. Contours

Question 1 Donner le mot de contour canonique du polyomino de la figure 1(b).

Question 2 Montrer que si u est le mot d'un contour d'un polyomino simplement connexe, il contient autant de g que de d, et autant de h que de b. Cette condition est-elle suffisante ? Pourquoi ?

Les mots de contour sont représentés par la classe *Contour* qui définit les quatre constantes D, H, G, B représentant les directions d, h, g, b et un tableau v d'entiers contenant le mot de contour.

```
class Contour {
  final static int D = 0, H = 1, G = 2, B = 3;
  int[ ] v;
  Contour (int n) { v = new int[n]; }
}
```

Question 3 Ecrire la fonction `testXY`(m) qui teste si le mot de contour m contient autant de g que de d et autant de h que de b. Quelle est sa complexité en temps par rapport à la longueur de m ?

Question 4 Ecrire la fonction `egal`(m, m') qui teste si les mots m et m' sont des mots de contours d'un même polyomino. Quelle est sa complexité en temps par rapport aux longueurs de m et de m' ?

On s'intéresse à présent aux polyominos simplement connexes de hauteur 1 ou 2, c'est-à-dire aux polyominos simplement connexes contenus dans la bande semi-infinie de hauteur deux définie par $A = \{(x, y) \mid 0 \leq y \leq 2,\ x \geq 0\}$ (voir exemple de la figure 1(a)). Soit L l'ensemble des mots de contours canoniques de tels polyominos.

Question 5 Dessiner tous les polyominos simplement connexes de surface au plus 3 dont le mot de contour canonique est dans L.

Question 6 Donner une grammaire algébrique (*context-free*) pour engendrer L.

Question 7 Donner une version non ambigue de cette grammaire, sans en démontrer formellement sa non-ambiguité.

Question 8 En déduire une fonction `imprimerL`(n) qui imprime tous les mots de contours canoniques des polyominos de hauteur 1 ou 2 de longueur inférieure ou égale à n avec une complexité $O(kn^2)$ en espace mémoire où k est le nombre de solutions. Cette fonction imprime-t-elle plusieurs fois le même mot de contour ?

Question 9 Donner des indications pour écrire la fonction `imprimerL`(n) avec une complexité $O(kn)$ en espace mémoire où k est le nombre de solutions.

Partie II. Génération aléatoire

L'objectif de cette partie est de construire des polyominos simplement connexes dont le complémentaire est 4-connexe. Pour cela, on utilise un algorithme probabiliste dont l'idée générale est la suivante. On part d'un point quelconque du plan ; on fait un chemin aléatoire dont chaque pas est choisi parmi les quatre directions possibles prises dans l'ensemble $\{d, h, g, b\}$ de manière aléatoire uniforme (en excluant la direction opposée à la dernière direction choisie). Dès que le chemin u ainsi construit crée un cycle, si ce cycle se termine sur le point de départ initial de u, il définit un polyomino simplement connexe (u correspond au parcours de la frontière du polyomino, dans le sens direct ou inverse) et on a fini ; sinon, u se décompose en un chemin v non vide suivi par un cycle, et on élimine le cycle avant de continuer la marche aléatoire à partir de v. Si cet algorithme termine, on veille à retourner le contour dans le sens direct. Dans cette construction, à tout moment, le chemin u construit ne peut posséder qu'au plus un cycle.

Comme la longueur maximale des contours de polyominos est inconnue dans cette partie, on représente à présent ces contours par des objets de la classe *ContourE*, c'est-à-dire par des listes de déplacements d, h, g, b. (Dans cette représentation, l'image miroir d'un contour est aussi représentable par un objet de la classe *ContourE*)

```
class ContourE {
  final static int D = 0, H = 1, G = 2, B = 3;
  Liste ch;
}

class Liste {
  int val; Liste suivant;
  Liste (int x, Liste a) { val = x; suivant = a; }
```

```
static Liste miroir (Liste a) {
  Liste r = null;
  while (a != null) {
    r = new Liste (a.val, r);
    a = a.suivant;
  }
  return r;
} } }
```

Question 10 Soit P un polyomino simplement connexe. On considère la valeur obtenue en faisant le tour de P dans le sens direct et en comptant $+1$ pour chaque virage d'un quart de tour vers la gauche et -1 pour chaque virage d'un quart de tour vers la droite. Montrer, par récurrence sur la surface de P, que la valeur totale lorsqu'on fait le tour de P une fois est exactement égale à 4.

Question 11 Soit u un chemin qui définit un polyomino simplement connexe. Écrire une fonction `estEnOrdreDirect`(u) qui vérifie si u est un parcours de la frontière dans le sens direct. Quelle est la complexité en temps de cette fonction ?

En Java, la classe `Random` possède la méthode `nextInt`(n) pour tout objet de sa classe qui retourne un nombre entier aléatoire entre 0 (inclus) et n (exclus). Ainsi pour générer un nombre dans $\{0, 1, 2, 3\}$, on écrit :

```
Random rand = new Random(); // en tête de la fonction
...
int d = rand.nextInt(4); // pour avoir un nouveau nombre aléatoire
...
```

Question 12 Écrire une fonction `genererUnContour`() qui génère un contour aléatoire d'un polyomino simplement connexe, selon l'algorithme indiqué au début de cette partie.

(Remarque : il est possible que le programme ne termine pas si le chemin ne revient pas à son point de départ. Cependant, on peut démontrer que la probabilité d'un tel événement est nulle.)

Partie III. Connexité simple

Soit N un entier fixé ($N > 0$). On souhaite décider si la région du plan $A = \{(x, y) \mid 0 \leq x, y \leq N\}$ contient un seul polyomino simplement connexe, comme sur la figure 2.

On représente le plan par une matrice de booléens *noir* telle que $noir[i, j] = true$ si et seulement si la région $\{(x, y) \mid i \leq x \leq i + 1, j \leq y \leq j + 1\}$ est contenue dans un polyomino. On suppose, par ailleurs, que la classe *Paire* suivante permet de manipuler des paires d'entiers.

```
class Region {
  boolean[ ][ ] noir;
  Region (int n) {
    noir = new boolean[n][n];
} }
```

```
class Paire {
  int x, y;
  Paire (int x0, int y0) {
    x = x0; y = y0;
} }
```

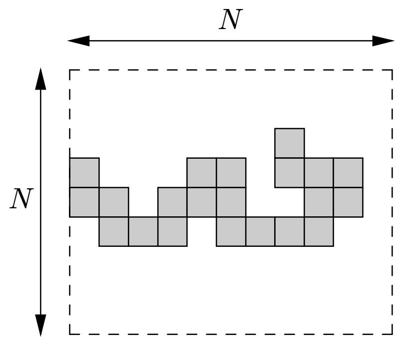


FIG. 2 – Région

Question 13 Écrire une fonction `trouverUnPoint`(r) qui renvoie la paire de coordonnées d'un point d'un polyomino contenu dans la région r. Cette fonction retourne `null` si ce point n'existe pas. Quelle est la complexité en temps de cette fonction ?

Question 14 Écrire une fonction `contientUnPolyomino`(r) qui retourne la valeur vraie si la région r contient un seul polyomino 4-connexe de surface maximale, et que ce polyomino soit simplement connexe. Quelle est la complexité en temps de cette fonction ? (Expliquez votre algorithme en termes simples de structures de données)

Une région peut $A = \{(x, y) \mid 0 \leq x, y \leq N\}$ peut à présent contenir plusieurs polyominos, certains sont simplement connexes, d'autres sont avec trous.

Question 15 Écrire une fonction `compterPolyominos`(r) qui compte le nombre de polyominos simplement connexes parmi les polyominos 4-connexes de surfaces maximales contenus dans la région r. (Expliquez à nouveau votre algorithme ; on pourra procéder de l'extérieur vers l'intérieur)

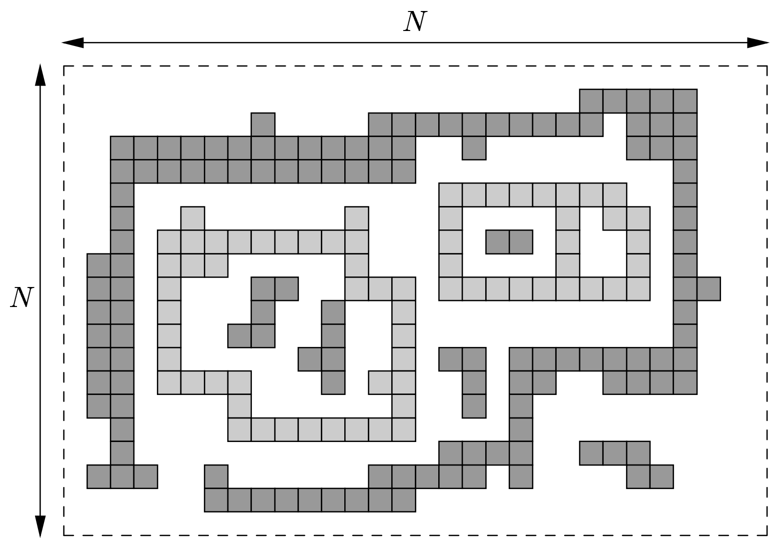


FIG. 3 – Région avec 6 polyominos simplement connexes de surfaces maximales (en couleur foncée)

1 Corrigé

Question 1 $db^2d^6h^5ghggbdbggbbdhdbbgggh^4dhggbbgbbdbb$

Question 2 Si (x, y) sont les coordonnées d'un point du contour, on revient au même point après un tour complet. Donc la somme totale des déplacements en X et Y est nulle.

Cette condition n'est pas suffisante. Contrexemples : *dg*, *hdbg*, *dhhdbggb*.

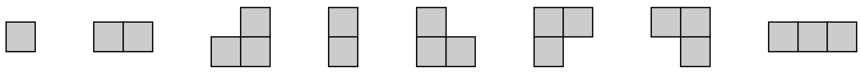
Question 3 Solution en $O(n)$ où n est la longueur de m.

```
static boolean testXY (Contour m) {
  int dx = 0, dy = 0;
  for (int i = 0; i < m.v.length; ++i)
    switch (m.v[i]) {
    case D: ++dx; break;
    case H: ++dy; break;
    case G: --dx; break;
    case B: --dy; break;
    }
  return dx == 0 && dy == 0;
}
```

Question 4 Solution en $O(n^2)$ où n est la longueur de m et de *m1*.

```
static boolean egal (Contour m, Contour m1) {
  int n = m.v.length, n1 = m1.v.length;
  boolean res = false;
  if (n == n1)
    for (int i = 0; !res && i < n; ++i) {
      res = true;
      for (int j = 0; res && j < n; ++j) {
        res = m1.v[j] == m.v[(i+j) % n];
    } }
  return res;
}
```

Question 5



Question 6 On décompose les polyominos selon la largeur les mots de L. Ce qui donne

$$\begin{array}{lllll}
L & \rightarrow & d\,L_1\,gb & \mid d\,L_2\,gbb & \mid d\,L_3\,gb \\
L_1 & \rightarrow & h & \mid d\,L_1\,g & \mid d\,L_2\,gb \\
L_2 & \rightarrow & hh & \mid d\,L_1\,gh & \mid hd\,L_3\,g & \mid d\,L_2\,g \\
L_3 & \rightarrow & h & \mid d\,L_3\,g & \mid bd\,L_2\,g
\end{array}$$

Question 7 Dans la décomposition précédente, on peut avoir deux fois les polyominos de hauteur 1. Il faut donc ne les faire apparaitre que dans une seule décomposition :

$$\begin{array}{lllll}
L & \rightarrow & d\,L_1\,gb & \mid d\,L_2\,gbb & \mid d\,L'_3\,gb \\
L_1 & \rightarrow & h & \mid d\,L_1\,g & \mid d\,L_2\,gb \\
L_2 & \rightarrow & hh & \mid d\,L_1\,gh & \mid hd\,L_3\,g & \mid d\,L_2\,g \\
L_3 & \rightarrow & h & \mid d\,L_3\,g & \mid bd\,L_2\,g \\
L'_3 & \rightarrow & & d\,L'_3\,g & \mid bd\,L_2\,g
\end{array}$$

Question 8 On suit la définition de la grammaire précédente ; il n'y a qu'une seule occurrence de chaque contour, sinon la grammaire serait ambigue. Ce programme prend une place en $O(kn^2)$ puisque chaque concaténation crée un nouvel espace mémoire. En fait seul $O(n^2)$ n'est nécessaire à tout moment.

```
static void imprimerL (int n) {
  if (n >= 4) iL1 ("d", n-3, "gb");
  if (n >= 6) iL2 ("d", n-4, "gbb");
  if (n >= 8) iL3bis ("d", n-3, "gb");
}

static void iL1 (String s, int n, String t) {
  if (n >= 1) iSol (s+"h"+t);
  if (n >= 3) iL1 (s+"d", n-2, "g"+t);
  if (n >= 5) iL2 (s+"d", n-3, "gb"+t);
}

static void iL3 (String s, int n, String t) {
  if (n >= 1) iSol (s+"h"+t);
  if (n >= 3) iL3 (s+"d", n-2, "g"+t);
  if (n >= 5) iL2 (s+"bd", n-3, "g"+t);
}

static void iL2 (String s, int n, String t) {
  if (n >= 2) iSol (s+"hh"+t);
  if (n >= 4) iL2 (s+"d", n-2, "g"+t);
  if (n >= 4) iL1 (s+"d", n-3, "gh"+t);
  if (n >= 4) iL3 (s+"hd", n-3, "g"+t);
}

static void iL3bis (String s, int n, String t) {
  if (n >= 9) iL3bis (s+"d", n-2, "g"+t);
  if (n >= 7) iL2 (s+"bd", n-3, "g"+t);
}

static void iSol (String s) { System.out.println (s); }
```

Question 9 Pour réduire la place mémoire, il suffit de passer un contour de taille n qu'on remplit graduellement pour le préfixe s et le suffixe t dans les fonctions précédents. Ainsi le programme suivant occuperait une taille $0(kn)$ (en fait $O(n)$ mémoire nécessaire à tout moment) :

```
static void imprimerL (int n) {
  Contour m = new Contour (n);
  if (n >= 4) {m.v[0] = D; m.v[n-2] = G; m.v[n-1] = B; iL1(m, 1, n-2);}
  if (n >= 6) {m.v[0] = D; m.v[n-3] = G; m.v[n-2] = m.v[n-1] = B; iL2(m, 1, n-3);}
  if (n >= 8) {m.v[0] = D; m.v[n-2] = G; m.v[n-1] = B; iL3bis(m, 1, n-2);}
}

static void impL1 (Contour m, int i, int j) {
  if (j - i >= 1) {m.v[i] = H; impSol (m, i+1, j);}
  if (j - i >= 3) {m.v[i] = D; m.v[j-1] = G; impL1 (m, i+1, j-1);}
  if (j - i >= 5) {m.v[i] = D; m.v[j-2] = G; m.v[j-1] = B; impL2 (m, i+1, j-2);}
}
...
```

Question 10 Vrai si P a une surface 1. Soit P_{n+1} de surface $n+1$ $(n > 1)$ qui a été obtenu à partir d'un P_n en rajoutant un carré élémentaire. On raisonne sur le nombre de cotés communs entre P_n et P_{n+1} et à chaque fois on indique le changement dans le contour et on vérifie l'invariance de la somme :

Cas avec 1 coté commun :
hhh devient *hdhgh*. Alors on a bien $0 = -1+1+1-1$
dhh devient *ddhgh*. Alors on a bien $1 = 0+1+1-1$
hhg devient *hdhgg*. Alors on a bien $1 = -1+1+1+0$
dhg devient *ddhgg*. Alors on a bien $2 = 0+1+1+0$

Cas avec 2 cotés communs :
hghg devient *hhgg*. Alors on a bien $1 = 0+1+0$
hghh devient *hhgh*. Alors on a bien $0 = 0+1+-1$
gghg devient *ghgg*. Alors on a bien $0 = 0+1+-1$
gghh devient *ghgh*. Alors on a bien $-1 = -1+1+-1$

Cas avec 3 cotés communs :
hghdh devient *hhh*. Alors on a bien $1+-1+-1+1 = 0$
gghdh devient *ghh*. Alors on a bien $0+-1+-1+1 = -1$
gghdd devient *ghd*. Alors on a bien $0+-1+-1+0 = -1+-1$
hghdd devient *hhd*. Alors on a bien $1+-1+-1+0 = 0+-1$

Question 11 La fonction précédente prend un temps $O(n)$ où n est la longueur de u.

```
static boolean estEnOrdreDirect (ContourE u) {
  int s = 0;
  for (Liste a = u.ch; a != null; a = a.suivant)
    if (a.suivant != null) {
      int delta = a.suivant.val - a.val;
      if (delta == -3) delta = 1;
      else if (delta == 3) delta = -1;
      s = s + delta;
    }
  return 3 <= s && s <= 5;
}
```

Question 12

```
static ContourE genererUnContour () {
  Random rand = new Random();
  Liste a = null, b;
  do {
    int d; do { d = rand.nextInt(4); }
    while (a != null && Math.abs (d - a.val) == 2);
    a = new Liste(d, a);
    b = a; a = eliminerCycle(a);
  } while (a != null);
  return mettreEnOrdreDirect(new ContourE(b));
}

static Liste eliminerCycle (Liste a) {
  int[ ] dx = {1, 0, -1, 0}, dy = {0, 1, 0, -1};
  int x = 0, y = 0; Liste b = a;
  do {
    x = x + dx[b.val]; y = y + dy[b.val];
```

```
    b = b.suivant;
  } while (b != null && (x != 0 || y != 0)) ;
  if (x == 0 && y == 0) a = b;
  return a;
}

static ContourE mettreEnOrdreDirect (ContourE u) {
  if (! estEnOrdreDirect(u))
    u.ch = Liste.miroir (u.ch);
  return u;
}
```

Question 13 La fonction suivante prend un temps $O(N^2)$.

```
Paire trouverUnPoint (Region r) {
  int n = r.noir[0].length;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      if (r.noir[i][j]) return new Paire (i, j);
  return null;
}
```

Question 14 On considère le graphe dont *noir* est la matrice d'adjacence. On fait *dfs* sur l'extérieur toujours non vide (on rajoute virtuellement une bande d'unité un autour de la région) ; puis on va chercher un point dans le polyomino ; on fait *dfs* sur le polyomino. Si enfin, on a visité tous les points de la région, la région contient bien un seul polyomino simplement connexe. Au total cette fonction est en $O(N^2)$.

```
static boolean contientUnPolyomino (Region r) {
  Paire p = trouverUnPoint(r);
  if (p == null) return false;
  else {
    int n = r.noir.length;
    boolean[ ][ ] vu = new boolean[n+2][n+2];
    int nNoirs = dfs (r, p.x, p.y, 0, vu);
    int nBlancs = dfs1 (r, -1, -1, 0, vu);
    return nNoirs + nBlancs == n * n + 4*(n + 1) ;
  }
}

static int dfs (Region r, int i, int j, int nc, boolean[ ][ ] vu) {
  int n = vu.length - 2;
  if (r.noir[i][j] && ! vu[i+1][j+1]) {
    vu[i+1][j+1] = true; ++nc;
    if (i > 0) nc = dfs (r, i-1, j, nc, vu);
    if (j > 0) nc = dfs (r, i, j-1, nc, vu);
    if (i < n-1) nc = dfs (r, i+1, j, nc, vu);
    if (j < n-1) nc = dfs (r, i, j+1, nc, vu);
  }
  return nc;
}

static int dfs1 (Region r, int i, int j, int nc, boolean[ ][ ] vuExt) {
  int n = vuExt.length - 2;
  if (-1 <= i && i <= n && -1 <= j && j <= n)
```

```
    if ((i == -1 || j == -1 || i == n || j == n ||
        !r.noir[i][j]) && ! vuExt[i+1][j+1]) {
      vuExt[i+1][j+1] = true; ++nc;
      for (int a = -1; a <= 1; ++a)
        for (int b = -1; b <= 1; ++b)
          nc = dfs1 (r, i+a, j+b, nc, vuExt);
    }
    return nc;
  }
```

Question 15 On considère à nouveau le graphe dont *noir* est la matrice d'adjacence. On explore les sommets du graphe par ordre lexicographique à partir du point $(-1, -1)$ dans la bande extérieure blanche. Si le premier point non marqué est blanc, on marque sa composante 8-connexe (non vide) ; si c'est un point noir, on marque sa composante 4-connexe, et si celle-ci ne touche pas de sommet blanc non marqué, c'est un nouveau polyomino simplement connexe ; et on recommence sur le premier non marqué.

```
static int compterPolyominos (Region r) {
    int n = r.noir.length;
    boolean[ ][ ] vu = new boolean[n+2][n+2];
    int nPols = 0;
    for (int i = -1; i <= n; ++i)
      for (int j = -1; j <= n; ++j)
        if (!vu[i+1][j+1]) {
          if (i == -1 || j == -1 || i == n || j == n || !r.noir[i][j])
            dfsBlanc (r, i, j, vu);
          else
            if (estSC (r, i, j, vu))
              ++nPols;
        }
    return nPols;
  }

  static boolean estSC (Region r, int i, int j, boolean[ ][ ] vu) {
    int n = vu.length - 2;
    boolean res = vu[i+1][j+1] || r.noir[i][j];
    if (r.noir[i][j] && ! vu[i+1][j+1]) {
      vu[i+1][j+1] = true;
      if (i > 0) res = estSC (r, i-1, j, vu) && res;
      if (j > 0) res = estSC (r, i, j-1, vu) && res;
      if (i < n-1) res = estSC (r, i+1, j, vu) && res;
      if (j < n-1) res = estSC (r, i, j+1, vu) && res;
    }
    return res;
  }

  static void dfsBlanc (Region r, int i, int j, boolean[ ][ ] vu) {
    int n = vu.length - 2;
    if (-1 <= i && i <= n && -1 <= j && j <= n)
      if ((i == -1 || j == -1 || i == n || j == n || !r.noir[i][j])
         && ! vu[i+1][j+1]) {
        vu[i+1][j+1] = true;
        for (int a = -1; a <= 1; ++a)
```

```
        for (int b = -1; b <= 1; ++b)
          dfsBlanc (r, i+a, j+b, vu);
} }
```