École Polytechnique

INF411

Les bases de la programmation et de l'algorithmique

Jean-Christophe Filliâtre

rebroussement

remplir avec les entiers de 1 à 9, chaque entier étant utilisé une seule fois par ligne, par colonne et par groupe 3×3

2							6	
				7	5		3	
	4	8		9		1		
			3					
3				1				9
					8			
		1		2		5	7	
	8		7	3				
	9							4

écrivons un programme pour chercher une solution, le cas échéant, ou déterminer qu'il n'y en a pas

- construire la solution **incrémentalement**
- s'interrompre dès qu'on peut déterminer qu'une solution partielle ne peut être complétée, et revenir sur ses pas

on appelle cela le **rebroussement** (en anglais **backtracking)**

Sudoku

1. choisir une case vide
2. tester successivement la valeur 1,. . .,9 pour cette case
   - si elle est compatible, recommencer en 1
3. échouer

la grille de Sudoku est un tableau (où 0 indique une case vide)

```
class Sudoku {
  private int[] grid = new int[81];
```

et on se donne les méthodes (avec c,c1,c2 $\in$ [0,80])

```
  int   row(int c) { return c / 9; }
  int   col(int c) { return c % 9; }
  int group(int c) { return 3 * (row(c) / 3) + col(c) / 3; }
  boolean sameZone(int c1, int c2) {
    return   row(c1) == row(c2)
        ||   col(c1) == col(c2)
        || group(c1) == group(c2);
  }
```

on veut s'arrêter dès qu'une case p entre en conflit avec une autre case c

```
boolean check(int p) {
  for (int c = 0; c < 81; c++)
    if (c != p && sameZone(p, c) &&
        this.grid[p] == this.grid[c])
      return false;
  return true;
}
```

(suppose ici que grid[p] n'est pas nul)

le cœur de l'algorithme de rebroussement est une méthode **récursive**

```
boolean solve() {
  ...
}
```

qui

- en entrée,
  - suppose que `grid` ne contient pas de contradiction
- en sortie,
  - renvoie `true` si `grid` a pu être complétée en une solution
  - renvoie `false` si ce n'est pas possible et `grid` est inchangée

```
boolean solve() {
  for (int c = 0; c < 81; c++)          // choisir une
    if (this.grid[c] == 0) {            // case c vide
      for (int v = 1; v <= 9; v++) {
        this.grid[c] = v;               // tester avec v
        if (check(c) && solve())
          return true;
      }
      this.grid[c] = 0;                 // rebroussement
      return false;
    }
  return true;                          // grille pleine
}
```

la paresse de l'opérateur `&&` est ici essentielle

```
if (check(c) && solve())
  ...
```

ainsi, `solve()` n'est appelée que si `check(c)` renvoie `true`

bien sûr, on aurait pu écrire

```
if (check(c))
  if (solve())
    ...
```

la méthode `solve` n'a pas essayé toutes les cases vides
mais uniquement la première case vide trouvée

car si une solution existe, alors il y aura une valeur pour cette case

tester chaque case vide dans `solve` serait une perte de temps considérable

ici, on trouve une solution très rapidement (120 ms)

2	7	3	4	8	1	9	6	5
9	1	6	2	7	5	4	3	8
5	4	8	6	9	3	1	2	7
8	5	9	3	4	7	6	1	2
3	6	7	5	1	2	8	4	9
1	2	4	9	6	8	7	5	3
4	3	1	8	2	9	5	7	6
6	8	5	7	3	4	2	9	1
7	9	2	1	5	6	3	8	4

appels à `solve`

au total, on a fait 142 256 appels à `solve`

c'est très peu au regard de l'espace **potentiel** de recherche

il y a 58 cases vides

avec 9 valeurs par case

$$9^{58} \approx 2{,}22 \times 10^{55}$$

même avec les contraintes initiales

$$4 \times 4 \times 3 \times \cdots \times 3 \approx 1{,}42 \times 10^{32}$$

2							6	
				7	5		3	
	4	8		9		1		
			3					
3				1				9
					8			
		1		2		5	7	
	8		7	3				
	9							4

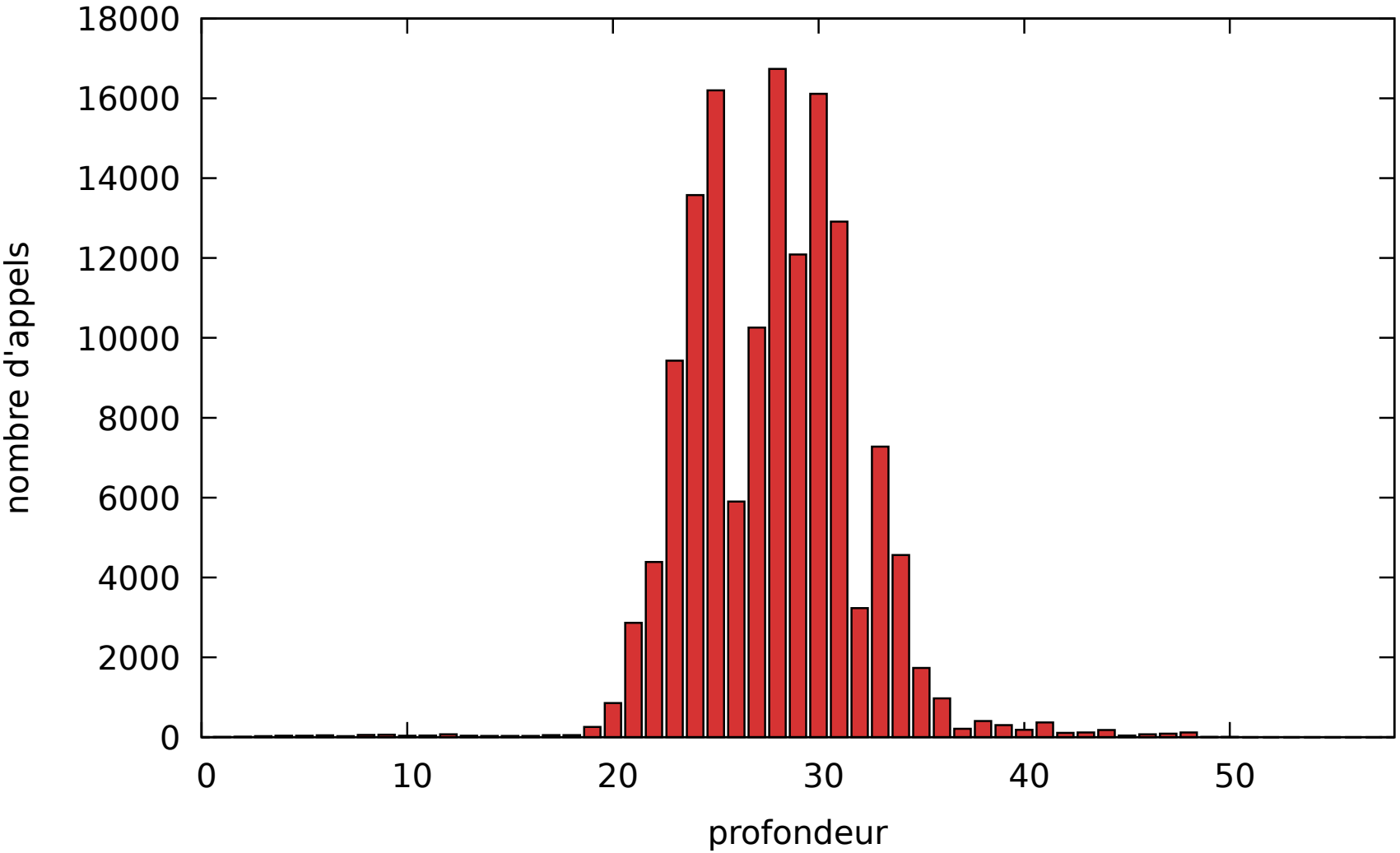
mais notre code réduit fortement cet espace de recherche au fur et à mesure de sa progression

il est intéressant d'observer le nombre d'appels faits **à chaque profondeur**

ici la profondeur varie entre 0 (premier appel) et 58 (appel sur une grille pleine)

profondeur	nombre d'appels
0	1
1	4
2	11
3	28
⋮	

répartition



et les autres solutions ?

et si on ne veut pas une solution, mais **toutes les solutions** ?
ou au moins les compter ?

il est facile d'adapter le code pour cela

```
int count = 0; // nombre de solutions trouvées
void count() {
  ...
}
```

cette fois la méthode ne renvoie rien et ne s'arrête plus à la première solution trouvée

néanmoins le code conserve la même structure

```
void count() {
  for (int c = 0; c < 81; c++)          // choisir une
    if (this.grid[c] == 0) {            // case c vide
      for (int v = 1; v <= 9; v++) {
        this.grid[c] = v;               // tester avec v
        if (check(c))
          count();
      }
      this.grid[c] = 0;                 // rebroussement
      return;
    }
  count++;                              // grille pleine
}
```

on trouve que notre grille n'a en fait qu'une seule solution

2							6	
				7	5		3	
	4	8		9		1		
			3					
3				1				9
					8			
		1		2		5	7	
	8		7	3				
	9							4

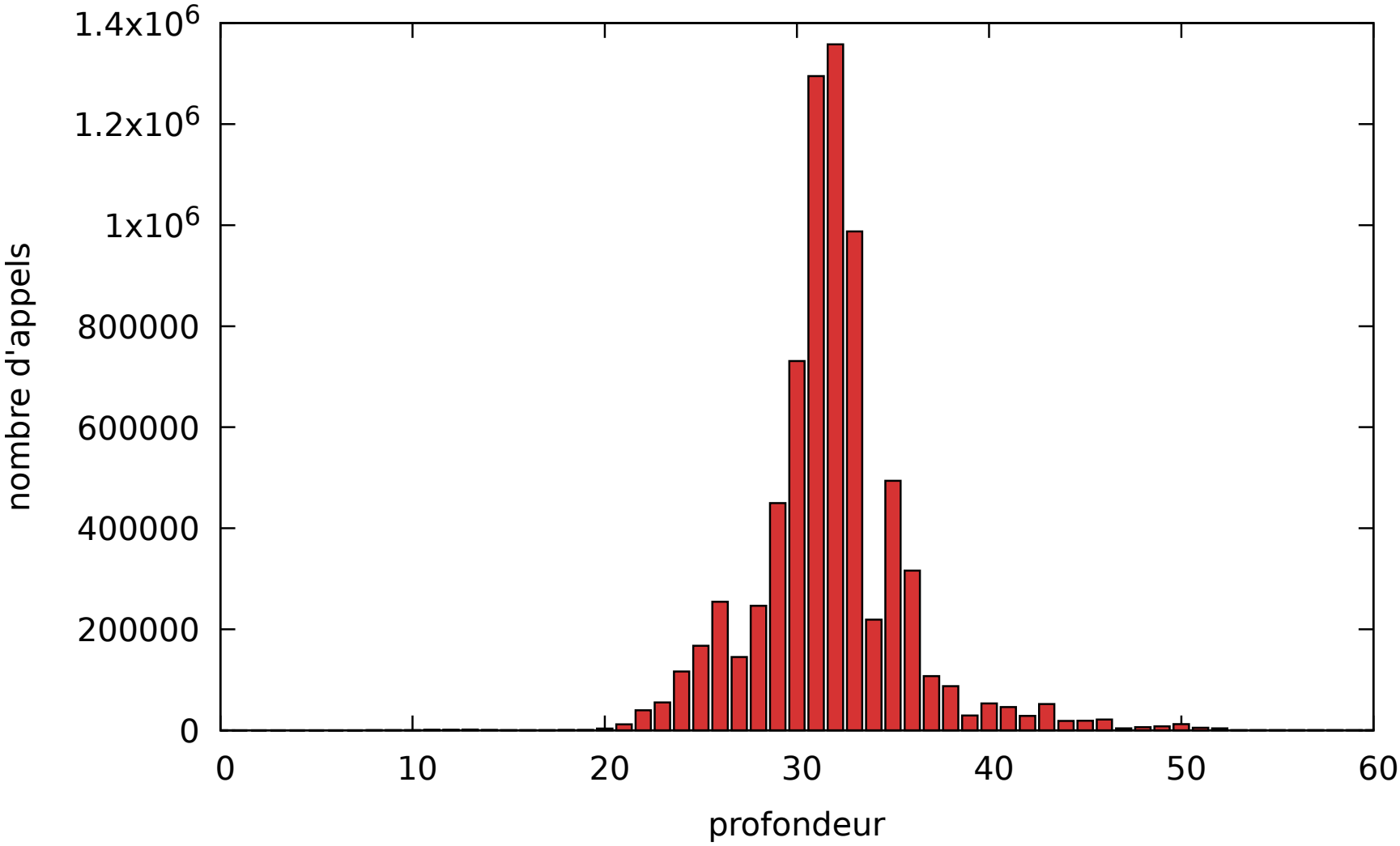
2	7	3	4	8	1	9	6	5
9	1	6	2	7	5	4	3	8
5	4	8	6	9	3	1	2	7
8	5	9	3	4	7	6	1	2
3	6	7	5	1	2	8	4	9
1	2	4	9	6	8	7	5	3
4	3	1	8	2	9	5	7	6
6	8	5	7	3	4	2	9	1
7	9	2	1	5	6	3	8	4

en à peine plus de temps (124 ms)
et un peu plus d'appels à `solve` (148 983)

sur une grille un peu moins contrainte

2							6	
				7			3	
	4	8		9		1		
			3					
3				1				
					8			
		1		2		5	7	
	8		7	3				
	9							4

on trouve 433 solutions en 5,5 secondes et 7 408 321 appels à `count`


1.4x10^6
1.2x10^6
1x10^6
800000
600000
400000
200000
0
nombre d'appels
0
10
20
30
40
50
60
profondeur

il est important de noter que le programme occupe **très peu de mémoire**
même s'il tourne longtemps et explore un très grand espace

en particulier, pas de risque de `StackOverflowError`
car jamais plus de 82 appels imbriqués

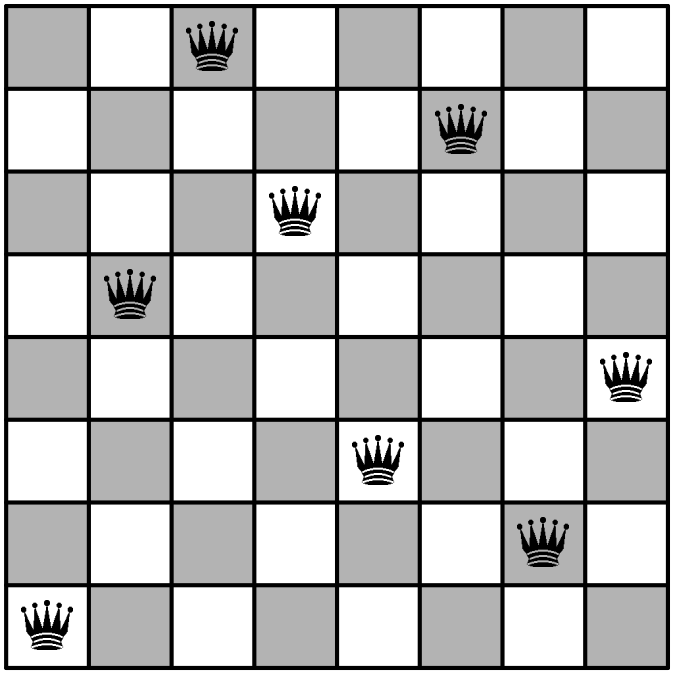
pour des Sudoku plus grands (16×16, 25×25, etc.),
il devient nécessaire d'améliorer l'efficacité de notre programme

par exemple en maintenant à chaque instant l'ensemble des valeurs possibles pour chaque case

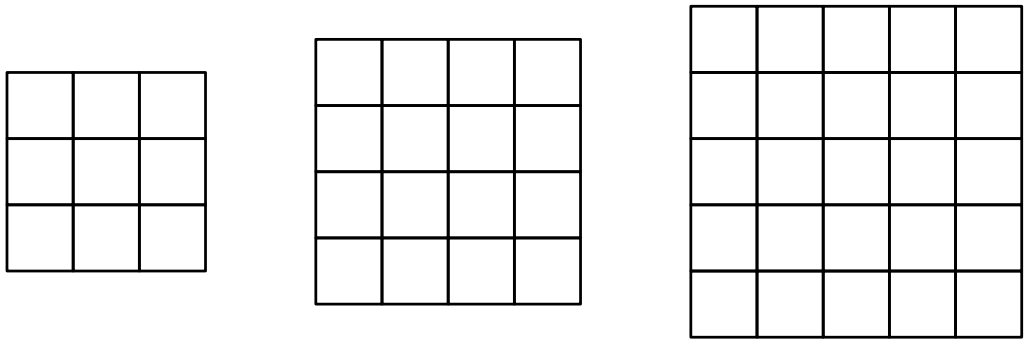
mais la structure du programme reste la même

d'autres problèmes

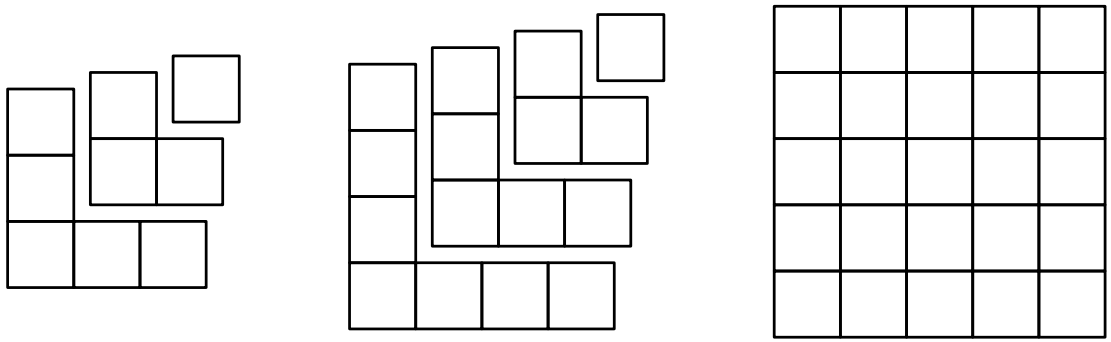
combien y a-t-il de façons de placer N reines sur un échiquier $N \times N$ sans qu'elles soient en prise ?



étant donné un triplet pythagoricien, par exemple (3,4,5),
peut-on paver le carré 5×5 avec les équerres des carrés 3×3 et 4×4 ?

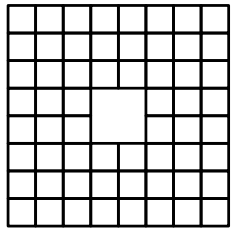


étant donné un triplet pythagoricien, par exemple (3,4,5),
peut-on paver le carré 5×5 avec les équerres des carrés 3×3 et 4×4 ?

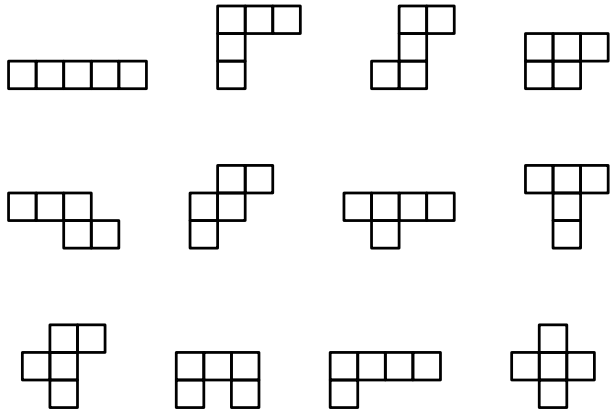


le problème de Scott (1958)

combien y a-t-il de façons de paver ces 60 cases



avec les douze pentaminos ?



bien entendu, on peut écrire un programme spécifique à chaque fois

mais y a-t-il un problème auquel on puisse se ramener souvent
dans l'espoir d'écrire un seul programme ?

la réponse est oui,
avec le problème de la **couverture exacte** (en anglais *exact cover*)

étant donnée une matrice de 0 et de 1

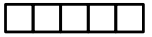
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

existe-t-il un sous-ensemble de lignes
couvrant chaque colonne exactement une fois ?

la matrice a

- 72 colonnes, une pour chaque pièce et une pour chaque case
- une ligne par façon de poser une pièce

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & \dots \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & \dots \\
\vdots & & & & & & & & & & & & & & & & & & & \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \dots \\
\vdots & & & & & & & & & & & & & & & & & & &
\end{pmatrix}$$

1568 lignes au total (48 lignes pour  , 96 pour  , etc.)

les liens dansants

les liens dansants [Knuth, 2000]

il s'agit d'un **algorithme** et d'une **structure de données** très efficaces pour résoudre le problème de la couverture exacte

il élimine progressivement des colonnes et des lignes de la matrice et procède par rebroussement en cas d'échec

`solve()`

1. si la matrice n'a plus de colonnes, on a une solution
2. sinon, choisir une colonne C arbitrairement
3. pour chaque ligne R couvrant C
   - 3.1 ajouter R à la solution
   - 3.2 supprimer toutes les colonnes couvertes par R (dont C) ainsi que toutes les lignes couvrant ces colonnes (dont R)
   - 3.3 appeler `solve()` récursivement
   - 3.4 annuler 3.1 et 3.2
4. échouer

choix de la colonne à couvrir

on observe (empiriquement) que choisir la colonne contenant le moins de 1 est une bonne stratégie

en particulier, si une colonne ne contient plus de 1, l'algorithme va échouer tout de suite

il est facile de maintenir le nombre de 1 restants de chaque colonne

déroulement de l'algorithme

	A	B	C	D	E	F	G
a	0	0	1	0	1	1	0
b	1	0	0	1	0	0	1
c	0	1	1	0	0	1	0
d	1	0	0	1	0	0	0
e	0	1	0	0	0	0	1
f	0	0	0	1	1	0	1

on choisit la colonne A

	A	B	C	D	E	F	G
a	0	0	1	0	1	1	0
b	1	0	0	1	0	0	1
c	0	1	1	0	0	1	0
d	1	0	0	1	0	0	0
e	0	1	0	0	0	0	1
f	0	0	0	1	1	0	1

on choisit la colonne A
on choisit la ligne b

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
⋆	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne A

on choisit la ligne b

on élimine A, D, G, b, d, e, f

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
⋆	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne B

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
⋆	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne B
on choisit la ligne c

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
⋆	b	1	0	0	1	0	0	1
⋆	c	0	1	1	0	0	1	0
	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne B
  on choisit la ligne c
    on élimine B, C, F, a, c

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
⋆	b	1	0	0	1	0	0	1
⋆	c	0	1	1	0	0	1	0
	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne E

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
⋆	b	1	0	0	1	0	0	1
⋆	c	0	1	1	0	0	1	0
	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

déroulement de l'algorithme

on choisit la colonne E
on échoue

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
$\star$	b	1	0	0	1	0	0	1
$\star$	c	0	1	1	0	0	1	0
	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

déroulement de l'algorithme

on choisit la colonne E

on échoue

on revient en arrière

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
$\star$	b	1	0	0	1	0	0	1
$\star$	c	0	1	1	0	0	1	0
	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne A
on choisit la ligne d

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne A
  on choisit la ligne d
    on élimine A, D, b, d, f

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne E

		A	B	C	D	E	F	G
	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne E
on choisit la ligne a

		A	B	C	D	E	F	G
⋆	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne E
  on choisit la ligne a
    on élimine C, E, F, a, c

		A	B	C	D	E	F	G
⋆	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne B

		A	B	C	D	E	F	G
⋆	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne B
on choisit la ligne e

		A	B	C	D	E	F	G
⋆	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
⋆	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on choisit la colonne B
on choisit la ligne e
on élimine B, G, e

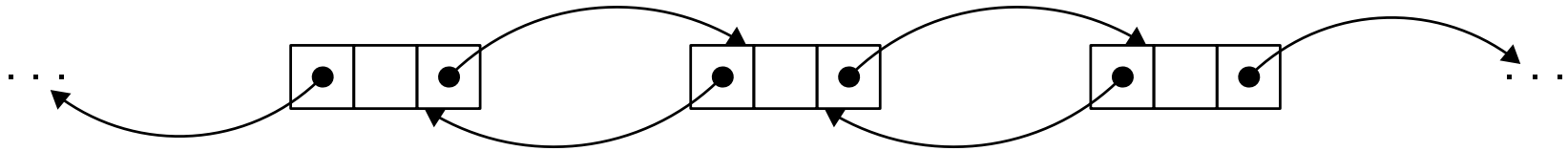
		A	B	C	D	E	F	G
⋆	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
⋆	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

on a une solution $\{a,d,e\}$

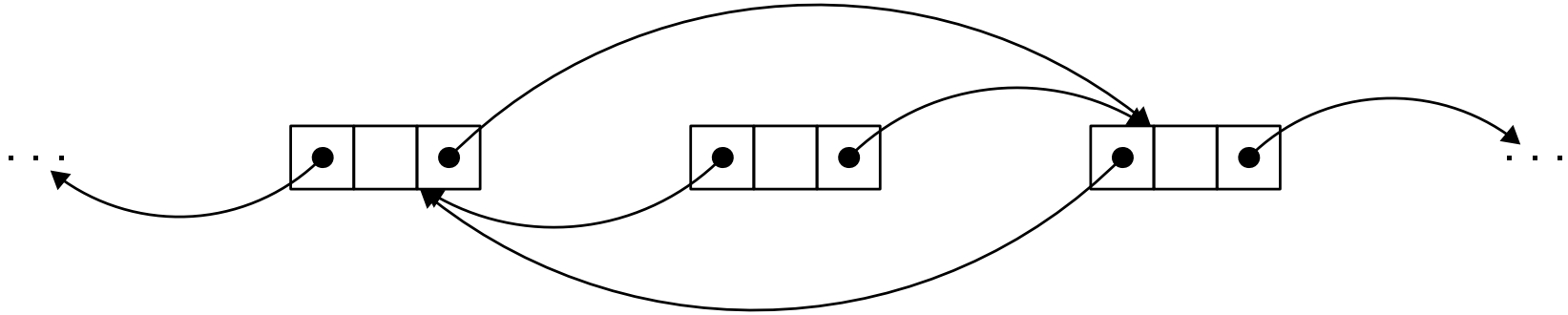
		A	B	C	D	E	F	G
⋆	a	0	0	1	0	1	1	0
	b	1	0	0	1	0	0	1
	c	0	1	1	0	0	1	0
⋆	d	1	0	0	1	0	0	0
⋆	e	0	1	0	0	0	0	1
	f	0	0	0	1	1	0	1

il faut trouver un moyen efficace de supprimer des lignes et des colonnes puis de les rétablir ensuite

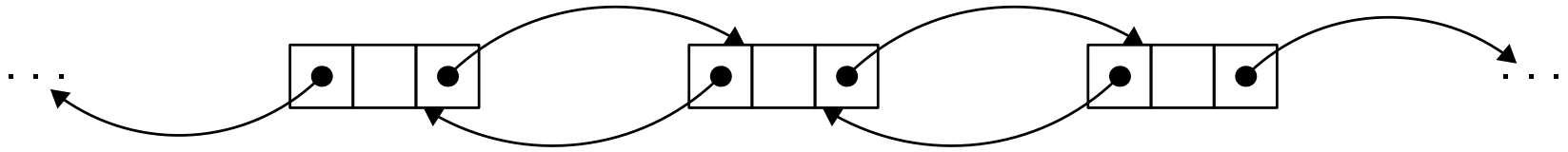
pour supprimer un élément d'une liste doublement chaînée



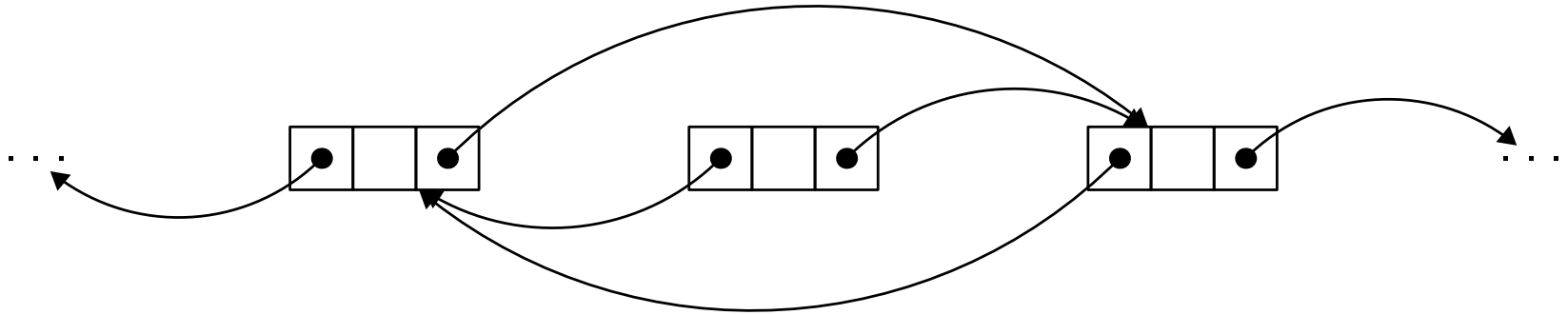
il suffit de faire pointer l'élément de gauche vers l'élément de droite
et vice-versa



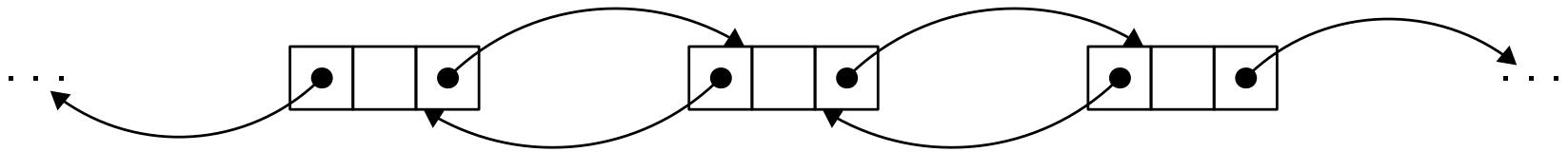
observation : l'élément supprimé conserve ses deux pointeurs
et il est donc facile de rétablir la situation initiale



```
this.left.right = this.right;
this.right.left = this.left;
```



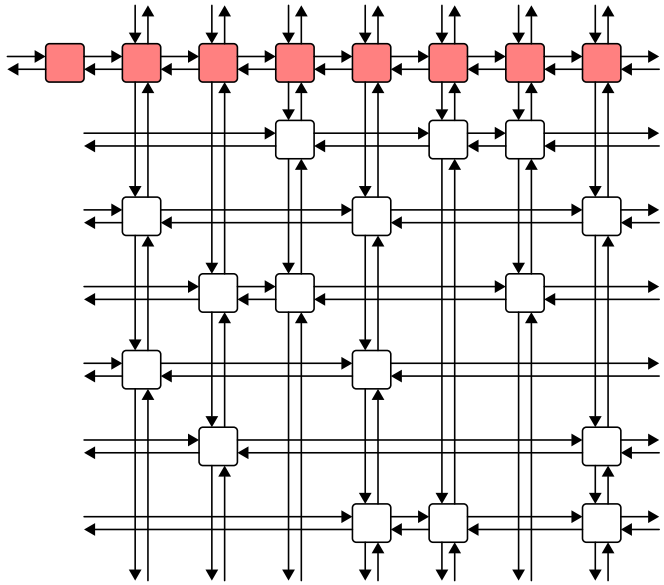
```
this.right.left = this;
this.left.right = this;
```



idée de la structure

relier entre eux tous les 1 de la matrice
dans des listes doublement chaînées circulaires horizontales et verticales

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



on ajoute un nœud par colonne et un entête (ici en rouge)

il devient très facile

- de supprimer une colonne (on l'enlève de la première liste)
- de parcourir les 1 d'une colonne ou d'une ligne (ils sont chaînés)
- de supprimer un 1 d'une ligne (on l'enlève de la liste verticale)
- de rétablir la situation précédente (avec l'observation faite plus haut, et en prenant soin de parcourir la liste **dans l'ordre inverse**)

les 0 de la matrice n'occupent aucune place en mémoire

la matrice peut être grosse mais très creuse

exemple : pour le problème de Scott plus haut, seulement 8% de 1

```
class Node {
  Node left, right, up, down;
  // pour un élément, sa colonne
  Node col;
  // pour une colonne, son nom et sa taille
  String name;
  int size;
```

couvrir la colonne c

```
void cover(Node c) {
  assert c.isColumn();
  c.removeLR();
  for (Node x = c.down; x != c; x = x.down)
    for (Node y = x.right; y != x; y = y.right) {
      y.removeUD();
      y.col.size--;
  }
}
```

dé-couvrir la colonne c

```
void uncover(Node c) {
  assert c.isColumn();
  for (Node x = c.up; x != c; x = x.up)
    for (Node y = x.left; y != x; y = y.left) {
      y.col.size++;
      y.restoreUD();
    }
  c.restoreLR();
}
```

on prend soin de procéder **en sens inverse**
(`up` à la place de `down` et `left` à la place de `right`)

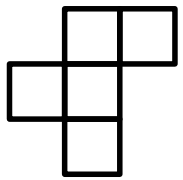
```
boolean solve() {
  if (header.right == header) return true;
  Node best = minColumn();
  cover(best);
  for (Node row = best.down; row != best; row = row.down) {
    sol.addLast(row);
    for (Node x = row.right; x != row; x = x.right)
      cover(x.col);
    if (solve()) return true;
    for (Node x = row.left; x != row; x = x.left)
      uncover(x.col);
    sol.removeLast();
  }
  uncover(best);
  return false;
}
```

application

on trouve ainsi 520 solutions au problème de Scott, très rapidement (en moins d'une seconde)

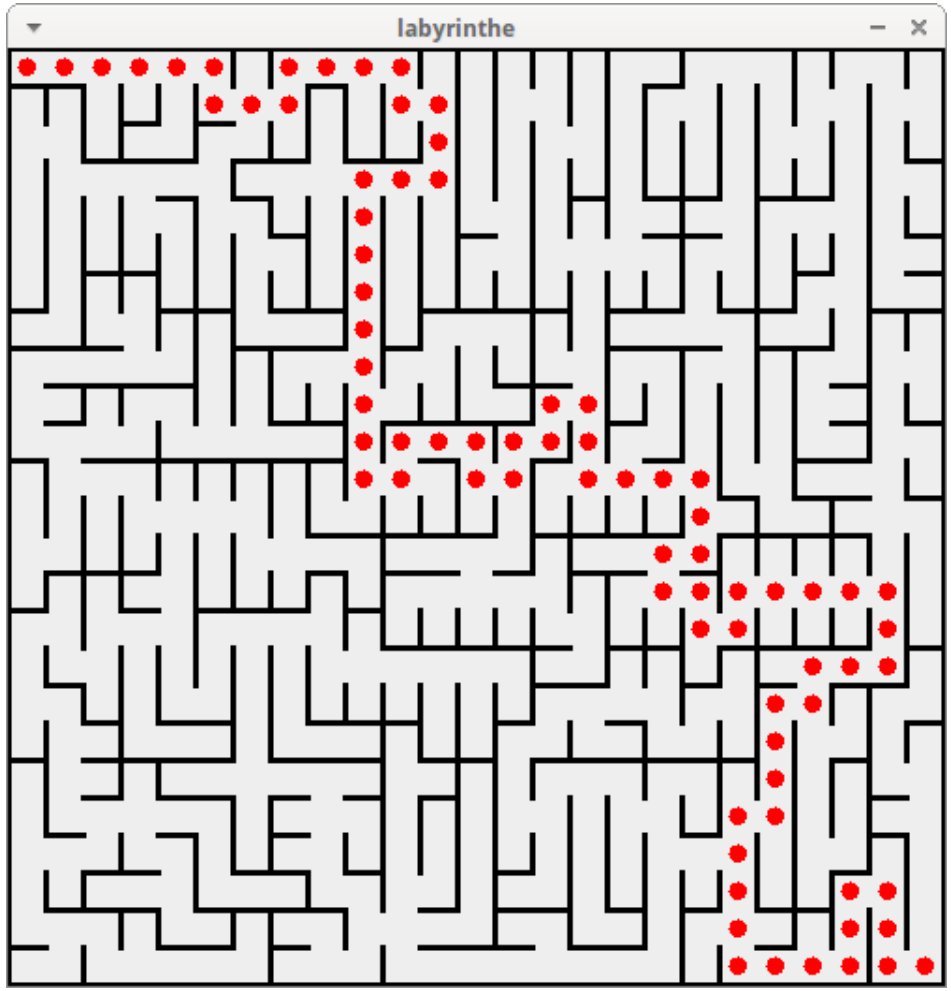
en réalité, il y a 65 solutions distinctes et 8 symétries à chaque fois

il est facile de se limiter aux 65 solutions, en plaçant dans une seule orientation une pièce comme



le TD de cette semaine

- trouver son chemin avec le rebroussement
- construire un labyrinthe avec du rebroussement
- construire un labyrinthe avec union-find, autrement



- **lire le poly**, chapitre 12

  12.1 Le problème du Sudoku

  12.2 Le problème des N reines

- **bloc 5** : arbres