

École Polytechnique

CSC_41011

Les bases de la programmation et de l'algorithmique


Jean-Christophe Filiâtre

classes disjointes

si vous avez aimé le TD de la semaine dernière



<http://projecteuler.net/>

- **mardi 7 octobre** (TD 7)
- comme un TD normal, mais
 -  sur les machines des salles info
 - pas d'accès à Internet (mais documentation de Java en local)

- **lire** le poly
 - faire quelques exercices (ils sont corrigés)
- **lire** les solutions des TD

- sauvegarder souvent (`Ctrl-S`)
- faire indenter le code par VSCode (`Ctrl-Shift-I`)
- avoir papier et crayon à côté de son clavier

rappel : les tables de hachage de Java

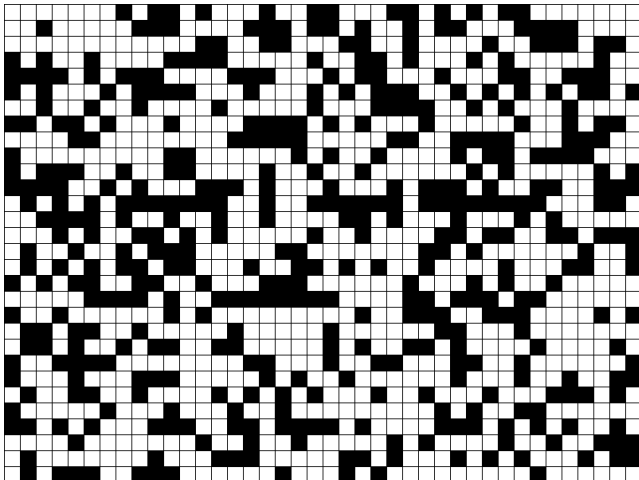
- `java.util.HashSet<E>`
ensemble d'éléments de type E
(add, contains, size, etc.)
- `java.util.HashMap<K, V>`
dictionnaire associant des valeurs de type V à des clés de type K
(put, get, size, etc.)

pour les utiliser, il faut **redéfinir** les méthodes `hashCode` et `equals` de la classe E/K

c'est déjà fait pour des classes comme `Integer` ou `String`

le problème d'aujourd'hui

la case en haut à gauche est-elle reliée à la case en bas à droite par des cases blanches uniquement ?



nous verrons plus tard (amphis 9 et 10) la notion de graphe, de chemin dans un graphe et des algorithmes pour trouver des chemins

mais il y a ici une solution plus élémentaire

l'idée consiste à construire progressivement une **partition** des cases, en un ensemble de **classes disjointes** de cases d'une même couleur connectées

il suffira alors de vérifier que la case en haut à gauche et la case en bas à droite appartiennent à la même classe

sans perte de généralité, cherchons à construire une partition de

$$\{0, 1, \dots, N - 1\}$$

(pour notre problème, $N = L \times H$ et $(x, y) \mapsto x \times H + y$)

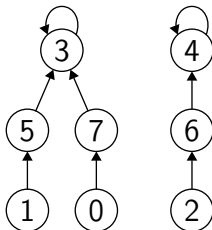
une interface possible est la suivante

```
class UnionFind {
    UnionFind(int n) // univers 0,1,...,n-1
    int find(int i) // représentant de la classe de i
    void union(int i, int j) // réunir les classes de i et j
}
```

le nom de la structure, **union-find**, vient de ces deux opérations

idée : **lier** les éléments entre eux, jusqu'au représentant de chaque classe

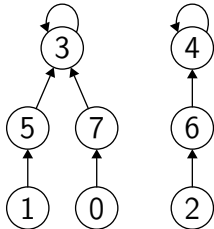
exemple : la partition $\{\{0, 1, 3, 5, 7\}, \{2, 4, 6\}\}$ peut être ainsi



avec les représentants 3 et 4 (arbitrairement)

on peut le faire avec un simple tableau

```
class UnionFind {  
    private int[] link;
```



0	1	2	3	4	5	6	7
7	5	6	3	4	3	4	3

initialement, on a N classes disjointes : $\{\{0\}, \{1\}, \dots, \{N - 1\}\}$

```
UnionFind(int n) {  
    if (n < 0) throw new IllegalArgumentException();  
    this.link = new int[n];  
    for (int i = 0; i < n; i++) this.link[i] = i;  
}
```

pour trouver le représentant, on suit les liaisons

```
int find(int i) {  
    int p = this.link[i];  
    if (p == i) return i;  
    return this.find(p);  
}
```

note : on aurait pu écrire une boucle `while` mais on verra plus loin

- qu'il n'y a pas de risque de provoquer `StackOverflowError`
- l'intérêt de la version récursive

pour faire l'union, on ajoute une liaison d'un représentant vers l'autre

```
void union(int i, int j) {  
    int ri = this.find(i);  
    int rj = this.find(j);  
    this.link[ri] = rj;    // on choisit rj arbitrairement  
}
```

note : sans effet si i et j sont déjà dans la même classe

comme expliqué plus haut, $N = L \times H$ et $(x, y) \mapsto x \times H + y$

```
int n = width * height;
UnionFind uf = new UnionFind(n);
```

pour chaque case

```
for (int x = 0; x < width; x++)
    for (int y = 0; y < height; y++) {
        int i = x * height + y;
```

on la relie à sa voisine de droite et/ou d'en dessous

```
    if (x < width-1 && cells[x][y] == cells[x + 1][y])
        uf.union(i, i + height);
    if (y < height-1 && cells[x][y] == cells[x][y + 1])
        uf.union(i, i + 1);
}
```

et on a la réponse à la question initiale

```
System.out.println(uf.find(0)==uf.find(n-1) ? "YES":"NO");
```

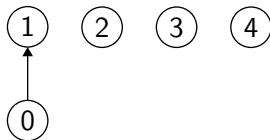
ici, le résultat est instantané avec une grille 40×30

mais qu'en serait-il avec une grille plus grande, par exemple $10^3 \times 10^3$?

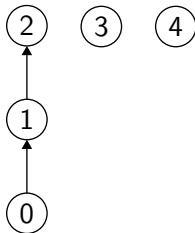
```
union(0, 1);  
union(0, 2);  
union(0, 3);  
union(0, 4);  
...
```



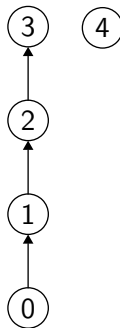
```
union(0, 1);  
union(0, 2);  
union(0, 3);  
union(0, 4);  
...
```



```
union(0, 1);  
union(0, 2);  
union(0, 3);  
union(0, 4);  
...
```



```
union(0, 1);  
union(0, 2);  
union(0, 3);  
union(0, 4);  
...
```



```
union(0, 1);  
union(0, 2);  
union(0, 3);  
union(0, 4);  
...
```



on peut se retrouver avec `find` et `union` coûtant $O(N)$

(et une solution quadratique en le nombre de cases pour notre problème)

cherchons à faire mieux...

dans `union`, on choisit pour représentant celui de “la plus grosse classe”

on appelle cela l'**union pondérée** (*weighted union*)

plutôt que de compter les éléments,

on mesure la **longueur maximale des chemins**, qu'on appelle le **rang**

un tableau stocke le rang de chaque classe

```
class UnionFind {  
    private int[] link;  
    private int[] rank; // uniquement pour les représentants
```

idée : une classe de rang k a des chemins de longueur au plus k

initialement, le rang de chaque classe est 0

```
UnionFind() {  
    ...  
    this.rank = new int[n];  
}
```

le code de union est modifié

```
void union(int i, int j) {
    int ri = this.find(i);
    int rj = this.find(j);
    if (ri == rj) return; // déjà dans la même classe
    if (this.rank[ri] < this.rank[rj])
        this.link[ri] = rj;
    else {
        this.link[rj] = ri;
        if (this.rank[ri] == this.rank[rj])
            this.rank[ri]++;
    }
}
```

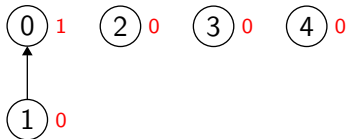
sur l'exemple précédent

```
uf.union(0, 1);  
uf.union(0, 2);  
uf.union(0, 3);  
uf.union(0, 4);  
...
```



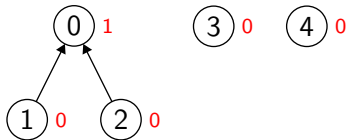
sur l'exemple précédent

```
uf.union(0, 1);  
uf.union(0, 2);  
uf.union(0, 3);  
uf.union(0, 4);  
...
```



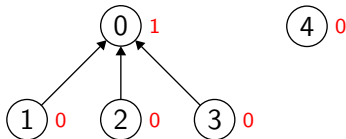
sur l'exemple précédent

```
uf.union(0, 1);  
uf.union(0, 2);  
uf.union(0, 3);  
uf.union(0, 4);  
...
```



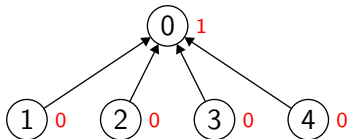
sur l'exemple précédent

```
uf.union(0, 1);  
uf.union(0, 2);  
uf.union(0, 3);  
uf.union(0, 4);  
...
```



sur l'exemple précédent

```
uf.union(0, 1);  
uf.union(0, 2);  
uf.union(0, 3);  
uf.union(0, 4);  
...
```



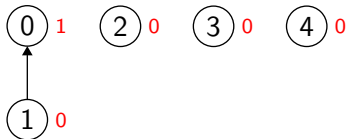
sur un autre exemple

```
uf.union(0, 1);  
uf.union(2, 3);  
uf.union(4, 3);  
uf.union(0, 2);  
...
```



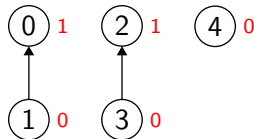
sur un autre exemple

```
uf.union(0, 1);  
uf.union(2, 3);  
uf.union(4, 3);  
uf.union(0, 2);  
...
```



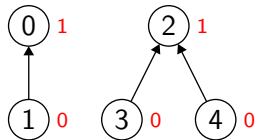
sur un autre exemple

```
uf.union(0, 1);  
uf.union(2, 3);  
uf.union(4, 3);  
uf.union(0, 2);  
...
```



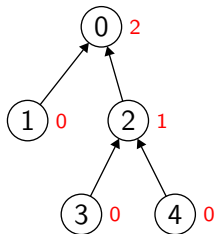
sur un autre exemple

```
uf.union(0, 1);  
uf.union(2, 3);  
uf.union(4, 3);  
uf.union(0, 2);  
...
```



sur un autre exemple

```
uf.union(0, 1);  
uf.union(2, 3);  
uf.union(4, 3);  
uf.union(0, 2);  
...
```



Propriété

une classe de rang k

- a des chemins de longueur maximale k
- possède au moins 2^k éléments

par récurrence sur le nombre d'appels à `union`

- c'est vrai initialement car toutes les classes sont de rang $k = 0$
- la dernière classe construite, de rang k , provient
 - d'une classe de rang k et d'une classe de rang $k' < k$
 - des nouveaux chemins de longueur maximale $k' + 1 \leq k$
 - au moins 2^k éléments provenant de la première classe
 - de deux classes de rang $k - 1$
 - des nouveaux chemins de longueur maximale $k - 1 + 1 = k$
 - au moins $2^{k-1} + 2^{k-1} = 2^k$ éléments



le coût de $\text{find}(i)$ est $O(k_i)$ où k_i est le rang de la classe de i

le coût de $\text{union}(i, j)$ est $O(\max(k_i, k_j))$
(deux appels à find , puis $O(1)$)

or le rang est au plus $\log(N)$, donc find et union sont en $O(\log(N))$

en particulier,
le nombre d'appels récursifs imbriqués de `find` est borné par $\log(N)$
on est donc assuré de ne pas faire déborder la pile d'appel
(`StackOverflowError`)

notre problème initial est donc maintenant résolu en $O(N \log(N))$

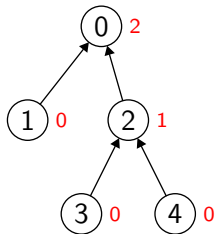
c'est tout à fait acceptable

cependant, on peut faire encore mieux

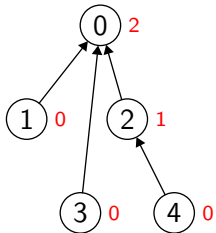
tout appel à `find(i)`, qui renvoie *r*, en profite pour lier directement *i* à *r*

on appelle cela la **compression de chemin**

```
uf.union(0, 1);  
uf.union(2, 3);  
uf.union(4, 3);  
uf.union(0, 2);  
int r = uf.find(3);  
...
```



```
uf.union(0, 1);  
uf.union(2, 3);  
uf.union(4, 3);  
uf.union(0, 2);  
int r = uf.find(3);  
...
```



une seule ligne à ajouter

```
int find(int i) {  
    int p = this.link[i];  
    if (p == i) return i;  
    int r = this.find(p);  
    this.link[i] = r;    // compression de chemin  
    return r;  
}
```

(d'où l'intérêt d'écrire find récursivement)

remarque : le rang n'est plus la longueur maximale des chemins, mais seulement un majorant

la propriété montrée plus haut devient

une classe de rang k

- a des chemins de longueur **au plus** k
- possède au moins 2^k éléments

avec les deux optimisations (rangs et compression de chemins),
on peut montrer que la complexité **amortie** de chaque opération est
 $O(\alpha(N))$, où α est l'inverse de la fonction d'Ackermann [Tarjan, 1979]

en pratique, $\alpha(N) < 5$ pour toute valeur réaliste de N

cette complexité est optimale [Fredman & Saks, 1989]

on a donc maintenant une solution **linéaire** à notre problème
(en pratique, pas en théorie)

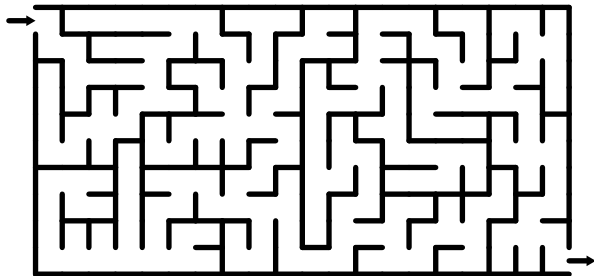
on peut très facilement étendre ce code pour maintenir

- le nombre de classes disjointes (ex 61 du poly)
- le nombre d'éléments de chaque classe
- la liste des éléments de chaque classe
- etc.

une fois la structure comprise, on l'adapte facilement aux besoins

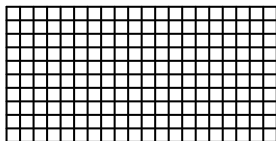
application

on va se servir de la structure **union-find** pour construire un **labyrinthe parfait**, c'est-à-dire un labyrinthe où deux cases sont toujours reliées par un et un seul chemin



(ici on distingue une case de départ et une case d'arrivée
mais la propriété est vraie pour toute paire de cases)

on part d'une grille où toutes les cases sont isolées par des murs



tant que toutes les cases ne sont pas reliées

- choisir une paire de cases adjacentes au hasard
- si elles ne sont pas encore reliées par un chemin, supprimer le mur entre ces deux cases

idée : maintenir les cases reliées dans une partition

(comme tout à l'heure, on prend $N = L \times H$ et $(x, y) \mapsto x \times H + y$)

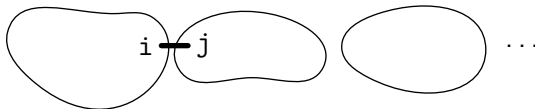
```
UnionFind uf = new UnionFind(N);
while (uf.numClasses() > 1) {
    // i,j = deux cases adjacentes au hasard
    ...
    if (uf.find(i) != uf.find(j)) {
        uf.union(i, j);
        // effacer le mur entre i et j
        ...
    }
}
```

on a l'invariant de boucle suivant

Propriété

deux cases sont reliées par un chemin si et seulement si elles appartiennent à la même classe, et elles sont alors reliées par un unique chemin

- vrai initialement (cases isolées et classes singletons)
- quand on fait `union(i, j)`, on relie les cases `i` et `j`



et l'invariant est bien préservé



on a donc bien un labyrinthe parfait une fois sorti de la boucle

on peut montrer que l'algorithme **termine avec probabilité 1**

mais on risque d'attendre tout de même longtemps...

vu qu'il est inutile de considérer deux fois la même paire de cases adjacentes, il suffit de les parcourir toutes dans un ordre aléatoire

question : comment mélanger correctement ?

par exemple les éléments d'un tableau

le mélange de Knuth (*Knuth shuffle*)

chaque élément i est échangé avec un élément $j \in [0, i]$

```
static<E> void shuffle(E[] a) {  
    for (int i = 1; i < a.length; i++) {  
        int j = (int)(Math.random() * (i + 1)); // j dans 0..i  
        swap(a, i, j);  
    }  
}
```

0	1	2	3	4	5	$j = 0$
1	0	2	3	4	5	$j = 2$
1	0	2	3	4	5	$j = 1$
1	3	2	0	4	5	$j = 4$
1	3	2	0	4	5	$j = 3$
1	3	2	5	4	0	

avant	<table border="1"><tr><td>x_0</td><td>x_1</td><td>x_2</td><td>\dots</td><td>x_{n-1}</td></tr></table>	x_0	x_1	x_2	\dots	x_{n-1}
x_0	x_1	x_2	\dots	x_{n-1}		
après	<table border="1"><tr><td>y_0</td><td>y_1</td><td>y_2</td><td>\dots</td><td>y_{n-1}</td></tr></table>	y_0	y_1	y_2	\dots	y_{n-1}
y_0	y_1	y_2	\dots	y_{n-1}		

la probabilité que $y_k = x_\ell$ vaut

$$P(y_k = x_\ell) = \frac{1}{n}$$

```
UnionFind uf = new UnionFind(N);  
// mettre toutes les adjacences dans un tableau a  
// ...  
shuffle(a);  
for (int k = 0; k < a.length; k++) {  
    // i,j = les deux cases de a[k]  
    ...  
    if (uf.find(i) == uf.find(j)) continue;  
    uf.union(i, j);  
    // effacer le mur entre i et j  
    ...  
}
```

(code sur la page du cours)

classes disjointes : variantes

- N n'est pas connu a priori ?
- les entiers ne sont pas consécutifs ?
- il ne s'agit pas d'entiers ?

- N n'est pas connu a priori?
⇒ tableau redimensionnable
- les entiers ne sont pas consécutifs ?
- il ne s'agit pas d'entiers ?

- N n'est pas connu a priori ?

⇒ tableau redimensionnable

- les entiers ne sont pas consécutifs ?
- il ne s'agit pas d'entiers ?

⇒ table de hachage

```
class HashUnionFind<E> {  
    private HashMap<E, E> link;  
    private HashMap<E, Integer> rank;  
    ...  
}
```

(voir aussi l'exercice 63 dans le poly)

application

- une égalité est-elle conséquence d'autres égalités :

$$x_1 = x_7 \wedge x_3 = x_8 \wedge \dots \Rightarrow x_4 = x_{17} ?$$

- même chose avec des fonctions non interprétées :

$$f(f(f(x))) = x \wedge f(f(f(f(f(x)))))) = x \Rightarrow f(x) = x ?$$

$$\frac{}{t = t}$$

$$\frac{t_1 = t_2}{t_2 = t_1}$$

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}$$

$$\frac{t_1 = t'_1 \quad \cdots \quad t_n = t'_n}{f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)}$$

très simple si on a uniquement des variables

$$x_1 = x_7 \wedge x_3 = x_8 \wedge \dots \Rightarrow x_4 = x_{17} ?$$

- on fait `union(i,j)` pour chaque égalité $x_i = x_j$ supposée
- on teste ensuite si $x_i = x_j$ avec `find(i)==find(j)`

plus complexe si on a aussi des symboles de fonctions

$$f(f(f(x))) = x \wedge f(f(f(f(f(x)))))) = x \Rightarrow f(x) = x ?$$

il faut utiliser alors un algorithme de **fermeture par congruence**
(*congruence closure algorithm*)

- on met tous les termes (et sous-termes) du problème dans une structure *union-find*

ici il s'agit de x , $f(x)$, $f^2(x)$, $f^3(x)$, $f^4(x)$ et $f^5(x)$

- pour chaque égalité $t_1 = t_2$ supposée, on fait `union(t_1 , t_2)`
- on fait `union($f(t)$, $f(t')$)` dès que `find(t)==find(t')`, jusqu'à saturation
- on teste enfin l'égalité cible $t_1 = t_2$ avec `find(t_1)==find(t_2)`

$$f^3(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x ?$$

initialement, un terme par classe

$$\{x\} \quad \{f(x)\} \quad \{f^2(x)\} \quad \{f^3(x)\} \quad \{f^4(x)\} \quad \{f^5(x)\}$$

$$f^3(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x ?$$

on fait $\text{union}(f^3(x), x)$ et $\text{union}(f^5(x), x)$

$$\{x, f^3(x), f^5(x)\} \quad \{f(x)\} \quad \{f^2(x)\} \quad \{f^4(x)\}$$

$$f^3(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x ?$$

congruence : puisque $x = f^3(x)$ alors $f(x) = f^4(x)$

$$\{x, f^3(x), f^5(x)\} \quad \{f(x)\} \quad \{f^2(x)\} \quad \{f^4(x)\}$$

$$f^3(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x ?$$

congruence : puisque $x = f^3(x)$ alors $f(x) = f^4(x)$

$$\{x, f^3(x), f^5(x)\} \quad \{f(x), f^4(x)\} \quad \{f^2(x)\}$$

$$f^3(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x ?$$

congruence : puisque $f(x) = f^4(x)$ alors $f^2(x) = f^5(x)$

$$\{x, f^3(x), f^5(x)\} \quad \{f(x), f^4(x)\} \quad \{f^2(x)\}$$

$$f^3(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x ?$$

congruence : puisque $f(x) = f^4(x)$ alors $f^2(x) = f^5(x)$

$$\{x, f^3(x), f^5(x), f^2(x)\} \quad \{f(x), f^4(x)\}$$

$$f^3(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x ?$$

congruence : puisque $f^2(x) = f^3(x)$ alors $f^3(x) = f^4(x)$

$$\{x, f^3(x), f^5(x), f^2(x)\} \quad \{f(x), f^4(x)\}$$

$$f^3(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x ?$$

congruence : puisque $f^2(x) = f^3(x)$ alors $f^3(x) = f^4(x)$

$$\{x, f^3(x), f^5(x), f^2(x), f^4(x), f(x)\}$$

la saturation est terminée et la réponse est donc oui

les termes (x , $f(x)$, etc.) sont représentés ainsi

```
class Term {  
    final String symb; // nom de fonction  
    final Term[] args; // arguments  
    ...  
}
```

une variable étant vue comme une fonction sans arguments
(tableau de longueur 0)

on munit cette classe de méthodes `hashCode` et `equals` adéquates
(cf le code sur la page du cours)

on utilise

- une structure *union-find*
- un ensemble de tous les (sous-)termes

```
class CongruenceClosure {  
    private HashUnionFind<Term> uf;  
    private      HashSet<Term> terms;
```

on remplit ces deux tables avec cette méthode

```
void add(Term t) {  
    if (terms.contains(t)) return; // déjà vu  
    terms.add(t);  
    uf.add(t);  
    for (Term s: t.args) // ajouter tous les sous-termes  
        add(s);  
}
```

on reconnaît là un schéma de mémorisation

il est facile de tester l'égalité de deux termes

```
uf.sameClass(t1, t2)
```

ou encore de deux tableaux de termes de même longueur

```
boolean checkEqArgs(Term[] l1, Term[] l2) {  
    for (int i = 0; i < l1.length; i++)  
        if (!uf.sameClass(l1[i], l2[i]))  
            return false;  
    return true;  
}
```

la fermeture est une boucle sur toutes les paires (t_1, t_2) , répétée tant qu'il y a changement

```
void cc() {  
    boolean change = true;  
    while (change) { ← jusqu'à saturation  
        change = false;  
        for (Term t1: this.terms)  
            for (Term t2: this.terms)  
                if (!uf.sameClass(t1, t2)  
                    && t1.symb.equals(t2.symb)  
                    && checkEqArgs(t1.args, t2.args)) {  
                    uf.union(t1, t2);  
                    change = true;  
                }  
        }  
    }  
}
```

il est inutile de considérer toute paire (t_1, t_2) ,
mais seulement les paires de la forme $(f(\dots), f(\dots))$ pour chaque f

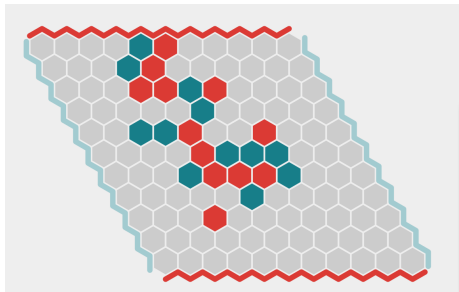
si on a beaucoup de termes, on peut avantageusement les indexer par
symbole de fonction

union-find, c'est

- une structure de données pour des classes disjointes (ou, si on préfère, des classes d'équivalence)
- une complexité amortie (quasi-)constante pour chaque opération

le jeu de Hex
(fr.wikipedia.org/wiki/Hex)

on gagne en construisant un chemin
d'un bord à l'autre



- **lire le poly**, chapitre 9

il y a des **exercices** dans le poly
suggestions : ex 61 page 122 et 63 page 124

- **bloc 4** : rebroussement