TP 7— Union-Find et minimisation d'automates

http://www.liafa.jussieu.fr/∼nath/tp7/tp7.pdf

Les questions sont les bienvenues et peuvent être envoyées à nathanael.fijalkow@gmail.com.

Ce deuxième TP de l'année fait suite au premier : notre objectif est maintenant de minimiser des automates déterministes. Pour cela, il faut savoir gérer des partitions, nous commençons donc par étudier l'algorithme Union-Find, avant de l'appliquer à la minimisation.

1 Algorithme Union-Find

On considère n objets (que nous appellerons $0, 1, \dots, n-1$) sur lesquels on définit des partitions. L'objectif est de construire une structure permettant d'effectuer les opérations suivantes :

- initialiser à la partition la plus fine où chaque classe est un singleton ;
- déterminer si deux éléments sont dans la même classe;
- modifier la relation pour fusionner les classes de deux éléments ;
- compter le nombre de classes de la partition ;

1.1 Une solution naïve

Donner une implémentation en mettant simplement à jour un vecteur v tel que `v.(x)` contient l'élément minimal de la classe d'équivalence de x.

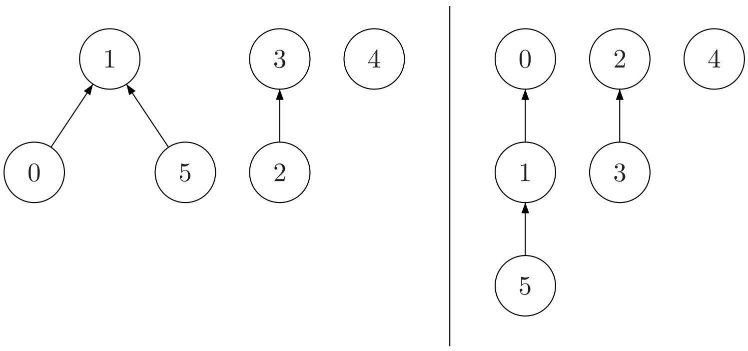
▷ **Question 1.** Écrire les fonctions suivantes :

```
initialise :  int -> int vect
compte :  int vect -> int
test_equ :  int vect -> int -> int -> bool
fusion :  int vect -> int -> int -> unit.
```

Quelle est la complexité de chacune de ces opérations ? ◁

1.2 Utiliser des arbres et des forêts

On peut représenter une partition par une forêt d'arbres. Par exemple la partition $\{0, 1, 5\}, \{2, 3\}, \{4\}$ pourra être représentée par l'une des forêts ci-dessous :



On représentera ce genre de forêt par un couple `(taille,pere)`, où `taille` est le nombre de parties de la partition, c'est-à-dire le nombre d'arbres de la forêt, et `pere.(x)` vaut `x` si `x` est la racine de son arbre, et la valeur de son père dans son arbre sinon.

Ainsi la structure utilisée est du type :

```
type structure = { mutable taille :  int ; pere :  int vect } ;;
```

▷ **Question 2.** Écrire pour cette nouvelle structure les fonctions suivantes :

```
initialise :  int -> int vect
compte :  int vect -> int
test_equ :  int vect -> int -> int -> bool
fusion :  int vect -> int -> int -> unit.
```

Quelle est la complexité de chacune de ces opérations ? ◃

1.3 Optimisation

Une première optimisation consiste à limiter la profondeur des arbres lorsque l'on effectue une fusion: pour cela, on maintient à jour un nouveau vecteur `poids`, tel que `poids.(x)` contienne la taille du sous-arbre enraciné en `x`.

La structure devient :

```
type structure = { mutable taille :  int ; pere :  int vect ; poids :  int vect } ;;
```

▹ **Question 3.** Modifier les fonctions pour utiliser la nouvelle structure. ◃

La seconde optimisation est la compression des chemins : à chaque fois qu'on calcule un représentant minimal pour une partie, on "remonte" les éléments considérés dans l'arbre, en les placant directement en dessous de la racine.

▹ **Question 4.** Modifier les fonctions en faisant de la compression des chemins. Que se passe t-il pour le tableau `poids` ? ◃

2 Minimisation d'automates déterministes

Les automates que l'on considère dans ce TP sont déterministes (pour chaque couple (état,lettre), il y a *au plus* une transition) et complet (pour chaque couple (état,lettre), il y a *au moins* une transition).

```
type automate =
  { taille :  int ;
    initial :  int ;
    transitions :  (char * int) list vect ;
    final :  bool vect } ;;
```

Pour minimiser un automate (déterministe et complet), on calcule l'équivalence de Nérode. Étant donné $\mathcal{A} = (Q = \{0, \ldots, n-1\}, q_0, \delta, F)$, c'est la relation d'équivalence sur Q définie par

$$p \sim q \Longleftrightarrow \forall w \in A^*,\ (p \cdot w \in L(\mathcal{A}) \Leftrightarrow q \cdot w \in L(\mathcal{A}))\ .$$

On la calcule par approximations successives : on définit les relations $\sim_k$ pour $k \in \mathbb{N}$ par

$$p \sim_k q \Longleftrightarrow \forall w \in A^{\leq k},\ (p \cdot w \in L(\mathcal{A}) \Leftrightarrow q \cdot w \in L(\mathcal{A}))\ .$$

$A^{\leq k}$ est l'ensemble des mots de longueur au plus k.

▹ **Question 5.** Montrer que $\sim\, = \,\sim_{n-2}$. ◃

▹ **Question 6.** Écrire une fonction `minimise` qui calcule l'automate minimal. ◃

3 Question difficile

▹ **Question 7.** Un autre cas d'école où l'algorithme Union-Find est intéressant est le calcul des composantes connexes (dans un graphe non-orienté) ou des composantes fortement connexes (dans un graphe orienté). Écrire deux fonctions `calcul_cc` et `calcul_cfc` qui calculent les partitions correspondantes à ces deux relations d'équivalences. ◃

▹ **Question 8.** Comment générer un labyrinthe en utilisant l'algorithme Union-Find ? ◃