I Jeu de Shannon

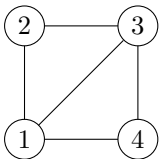
Soit $G = (V, E)$ un graphe non orienté.

1. Donner une condition équivalente simple pour que G contienne un arbre couvrant. Comment trouver un arbre couvrant algorithmiquement ?

   Solution : G contient un arbre couvrant si et seulement si G est connexe. Pour le déterminer, on peut utiliser un algorithme de parcours en profondeur ou largeur.
   Kruskal n'est pas vraiment pertinent ici.

Si T_1 et T_2 sont deux arbres couvrants de G, on dit qu'ils sont disjoints s'ils n'ont aucune arête en commun.

2. Le graphe suivant possède t-il deux arbres couvrants disjoints ?



   Solution : Non car 2 arbres couvrants disjoints auraient 6 arêtes au total, alors qu'il n'y en a que 5.

3. Soient T_1 et T_2 deux arbres couvrants de G et e_1 une arête de T_1. Montrer qu'il existe une arête e_2 de T_2 telle que $T_1 - e_1 + e_2$ (le graphe obtenu à partir de T_1 en enlevant e_1 et en ajoutant e_2) soit un arbre couvrant de G.

   Solution : Soit $T_1' = T_1 - e_1$. T_1' n'est pas connexe donc contient deux composantes connexes C_1 et C_2. T_2 est connexe donc contient une arête e_2 reliant C_1 et C_2. $T_1 - e_1 + e_2$ est alors un arbre couvrant de G.

4. Soit T un arbre couvrant de G et e une arête de T. On contracte e dans T et G, c'est-à-dire qu'on supprime e et on identifie ses deux extrémités, pour obtenir T' et G'. Montrer que T' est un arbre couvrant de G'.

Si P est une partition de V, on note $|P|$ son cardinal et $\|P\|$ le nombre d'arêtes de G dont les deux extrémités sont dans des ensembles différents de P.

On s'intéresse maintenant au théorème suivant :

Théorème de Tutte

Soit $k \in \mathbb{N}^*$. G possède k arbres couvrants disjoints si et seulement si, pour toute partition P de V, $\|P\| \geq k(|P| - 1)$.

5. En admettant le théorème de Tutte, montrer que le problème suivant appartient à NP.

   co-PACKING-TREES

   Entrée : Un graphe $G = (V, E)$ et un entier k.
   Question : est-il faux que G possède k arbres couvrants disjoints ?

6. Montrer le théorème de Tutte pour $k = 1$.

   Solution :

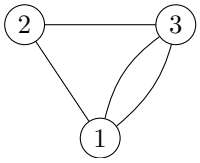
   - Supposons que, pour toute partition P de V, $\|P\| \geq |P| - 1$.
     Supposons par l'absurde que G ne soit pas connexe. Alors G possède deux composantes connexes C_1 et C_2. Soit $P = \{C_1, C_2\}$. Alors $\|P\| = 0$ et $|P| = 2$, ce qui contredit l'hypothèse : absurde.
     Donc G est connexe.
   - Supposons que G possède un arbre couvrant T et considérons une partition $P = \{C_1, C_2, \ldots, C_k\}$ de V.
     Soit $v \in V$ et C_i l'ensemble de P contenant v. On enracine T en v, en orientant les arêtes du père vers le fils. Alors chaque ensemble de P à part C_i possède un arc entrant (car T est connexe). Donc $\|P\| \geq k - 1$.

7. Montrer le sens direct du théorème de Tutte : si G possède k arbres couvrants disjoints, alors $\|P\| \geq k(|P| - 1)$.

   Solution : On reprend la même démonstration que le sens direct ci-dessus.

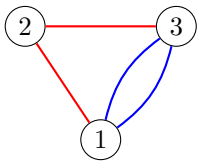
On considère un jeu avec un graphe $G = (V, E)$ non orienté qui peut posséder plusieurs arêtes entre deux sommets.
Deux joueurs A et B, où A commence, choisissent alternativement une arête de G non encore choisie. Si, à un moment de la partie, les arêtes choisies par B forment un arbre couvrant de G alors B gagne. Sinon, A gagne.

8. Indiquer si B a une stratégie gagnante si G est le graphe ci-dessous.



Solution : B a une stratégie gagnante, en choisissant une arête adjacente à 2 puis la dernière arête restante qui lui assure d'avoir un arbre couvrant.

Solution : L'attracteur de A est la situation où les arêtes $\{1,2\}$ et $\{2,3\}$ ont été choisies par A :



Dans la suite, on veut montrer que B possède une stratégie gagnante si et seulement si G possède deux arbres couvrants disjoints. Pour cela, on admet le théorème de Tutte.

9. Supposons qu'il existe une partition P de V telle que $\|P\| < 2(|P|-1)$. Montrer que A a une stratégie gagnante.

   Solution : Il suffit que A choisisse toujours une arête entre deux ensembles différents de P. Comme A commence, celui lui permet de sélectionner au moins la moitié de ces arêtes. Donc B en sélectionne $< |P|-1$, ce qui l'empêche de former un arbre couvrant.

10. Supposons qu'il existe deux arbres couvrants disjoints T_1 et T_2 dans G. Montrer que B a une stratégie gagnante.
    On pourra raisonner par récurrence sur $|V|$.

II Sous-graphe le plus dense

Soit $G = (V, E)$ un graphe non orienté à n sommets et p arêtes. Pour $S \subseteq V$, on définit la fonction de densité par :

$$\rho(S) = \frac{|E(S)|}{|S|}$$

où $E(S)$ est l'ensemble des arêtes de G ayant leurs deux extrémités dans S.

1. Quelles sont les valeurs minimum et maximum de $\rho(S)$, en fonction de $|S|$?

> Solution : Les minimum est $\rho(S) = 0$ quand S n'a aucune arête et le maximum est $\rho(S) = \frac{|S|(|S|-1)}{2}$ quand S est un sous-graphe complet.

2. Quel est le lien entre $\rho(S)$ et le degré moyen des sommets dans S ?

> Solution : $\rho(S) = \frac{|E(S)|}{|S|} = \frac{\sum_{u \in S} \deg(u)}{2|S|}$. $\rho(S)$ est donc égal à la moitié du degré moyen des sommets dans S.

On s'intéresse aux problèmes suivants :

DENSEST

Entrée : un graphe $G = (V, E)$.
Sortie : un ensemble $S \subseteq V$ tel que $\rho(S)$ soit maximum.

DENSEST-DEC

Entrée : un graphe $G = (V, E)$, un entier k et un réel α.
Sortie : existe t-il un ensemble $S \subseteq V$ tel que $|S| = k$ et $\rho(S) \geq \alpha$?

CLIQUE-DEC

Entrée : un graphe $G = (V, E)$ et un entier k.
Sortie : existe t-il un ensemble $S \subseteq V$ tel que $|S| = k$ et tous les sommets de S sont adjacents ($\forall u, v \in S, \{u, v\} \in E$) ?

3. En admettant que CLIQUE-DEC est NP-complet, montrer que DENSEST-DEC est NP-complet.

> Solution :
>
> - DENSEST-DEC $\in$ NP : on peut vérifier en temps polynomial qu'un ensemble S est de taille k et que $\rho(S) \geq \alpha$.
> - DENSEST-DEC $\geq$ CLIQUE-DEC : soit $G = (V, E)$ un graphe et k un entier. On pose $\alpha = \frac{k(k-1)}{2}$. Alors G a un sous-graphe complet de taille k si et seulement si G a un sous-graphe S de taille k tel que $\rho(S) \geq \alpha$.

On propose un algorithme glouton pour DENSEST :

- Itérativement retirer un sommet de degré minimum (ainsi que tous les sommets adjacents) jusqu'à ce qu'il n'y ait plus de sommet.
- À chacune de ces itérations, calculer la valeur de ρ et conserver le maximum.

4. Expliquer comment on pourrait implémenter cet algorithme en complexité temporelle $O(n + p)$.

> Solution : Pour obtenir $O(n+p)$ en moyenne, on peut utiliser un tableau t tel que $t[i]$ soit une liste doublement chaînée des sommets de degré i. On conserve aussi en mémoire le i_{min} minimum tel que $t[i_{min}]$ soit non-vide et, pour chaque sommet, un pointeur vers sa postition dans sa liste chaînée de t.
> À chaque itération, on retire un sommet v de $t[i_{min}]$ et on met à jour les voisins de v (en les retirant de $t[j]$ et en les ajoutant à $t[j-1]$). On met aussi à jour la densité ($\frac{|E(S)|}{|S|}$ est remplacé par $\frac{|E(S)| - \deg(v)}{|S| - 1}$) et i_min.
> Comme l'ajout et la suppression d'un élément dans une liste doublement chaînée se fait en temps constant, la complexité est bien $O(n + p)$.

Soit S^* tel que $\rho(S^*)$ soit maximum, $v^* \in S^*$ le premier sommet de S^* retiré par l'algorithme glouton et S' l'ensemble des sommets restants juste avant de retirer v^*.

5. Montrer que $\rho(S') \geq \frac{\deg_{S'}(v^*)}{2}$, où $\deg_{S'}(v^*)$ est le degré de v^* dans S'.

   Solution : Par choix de l'algorithme glouton : $\forall v \in S', \deg_{S'}(v) \geq \deg_{S'}(v^*)$. Donc :

   $$\rho(S') = \frac{|E(S')|}{|S'|} = \frac{\sum_{v \in S'} \deg_{S'}(v^*)}{2|S'|} \geq \frac{\deg_{S'}(v^*)|S'|}{2|S'|} \geq \frac{\deg_{S'}(v^*)}{2}$$

6. Justifier que $\rho(S^*) \geq \rho(S^* \setminus \{v^*\})$.

   Solution : Par optimalité de S^*.

7. En déduire que $\deg_{S^*}(v^*) \geq \rho(S^*)$.

   Solution : En utilisant $\rho(S^* \setminus \{v^*\}) = \frac{|E(S^*)| - \deg_{S^*}(v^*)}{|S^*| - 1}$ et la question 6, on obtient $\deg_{S^*}(v^*) \geq \rho(S^*)$.

8. En déduire que l'algorithme glouton est une 2-approximation pour DENSEST.

   Solution :

   $$\rho(S') \overset{5}{\geq} \frac{\deg_{S'}(v^*)}{2} \overset{(*)}{\geq} \frac{\deg_{S^*}(v^*)}{2} \overset{7}{\geq} \frac{\rho(S^*)}{2}$$

   Où $(*)$ vient de $S^* \subseteq S'$.

III Dominant

Soit $G = (V, E)$ un graphe. Un ensemble dominant de G est un sous-ensemble D de V tel que tout sommet de V est soit dans D, soit adjacent à un sommet de D.
On note $d(G)$ la taille d'un plus petit ensemble dominant de G.

1. Calculer $d(G)$ si G est un chemin à n sommets.

> Solution : Il existe un chemin dominant à $\left\lceil \frac{n}{3} \right\rceil$ sommets :
>
> 
>
> Comme chaque sommet peut dominer au plus 3 sommets, k sommets peuvent dominer au plus $3k$ sommets.
> Ainsi, un ensemble dominant à k sommets doit vérifier $3k \geq n$ et donc $k \geq \left\lceil \frac{n}{3} \right\rceil$.

2. On suppose que G est connexe et contient au moins 2 sommets. Montrer que $d(G) \leq \left\lfloor \frac{n}{2} \right\rfloor$.

> Solution : Soit T un arbre couvrant de G et v un sommet.
> On considère les ensembles D_0 et D_1 de sommets à distance paire et impaire de v dans T.
> D_0 et D_1 sont des ensembles dominants de G car tout sommet à distance paire est adjacent à un sommet à distance impaire (on utilise le fait que G pour cela) et réciproquement.
> Et $D_0 \sqcup D_1 = V$ donc D_0 ou D_1 est de taille au plus $\frac{n}{2}$.

3. On suppose que G ne contient pas de sommet isolé (sommet de degré 0). Montrer que $d(G) \leq \left\lfloor \frac{n}{2} \right\rfloor$.

> Solution : On peut appliquer la question précédente sur chaque composante connexe de G.

Une couverture par sommets de G est un sous-ensemble C de V tel que toute arête de G ait au moins une extrémité dans C.
On s'intéresse aux problèmes suivants :

DOMINANT

Entrée : Un graphe $G = (V, E)$ et un entier k.
Sortie : G possède-t-il un ensemble dominant de taille k ?

COUVERTURE

Entrée : Un graphe $G = (V, E)$ et un entier k.
Sortie : G possède-t-il une couverture par sommets de taille k ?

4. Soit G un graphe sans sommet isolé. Est-ce qu'une couverture par sommets est un ensemble dominant ? Et réciproquement ?

> Solution : Soit C une couverture par sommets de G. Si v est un sommet de G alors v est adjacent à une arête couverte par C donc v est adjacent à un sommet de C. Donc C est un ensemble dominant de G.
> Par contre, si G est un triangle (cycle de longueur 3) et v un sommet de G alors $\{v\}$ est un ensemble dominant de G mais pas une couverture par sommets.

5. On admet que COUVERTURE est NP-complet. Montrer que DOMINANT est NP-complet.

> Solution : DOMINANT est dans NP car on peut vérifier en temps polynomial qu'un ensemble de sommets est dominant.
> Soit G, k une instance de COUVERTURE. On note n_i le nombre de sommets isolés de G.
> On construit G' à partir de G en ajoutant un sommet v_e à chaque arête $e = \{u, v\}$ de G et en ajoutant une arête entre v_e et u et une arête entre v_e et v.
> Supposons que G possède une couverture par sommets C de taille k et considérons D obtenu à partir de C en ajoutant les sommets isolés de G. Alors D est un ensemble dominant de G' de taille $k + n_i$.
> Supposons inversement que G' possède un ensemble dominant D de taille k'. Soit C obtenu à partir de D en remplaçant chaque sommet de la forme v_e par l'une des extrémités de e et en enlevant les sommets isolés.
> Pour chaque arête e de G, il y a au moins un sommet parmi u, v et e dans D. Donc u ou v est dans C, ce qui permet

de conclure que C est une couverture par sommets de G de taille $k' - n_i$.
G possède donc une couverture par sommets de k si et seulement si G' possède un ensemble dominant de taille $k + n_i$.
Nous avons donc construit une réduction polynomiale de COUVERTURE vers DOMINANT, ce qui permet de conclure que DOMINANT est NP-complet.

6. Décrire un algorithme efficace pour résoudre DOMINANT si G est un arbre. L'implémenter en OCaml.

Solution : fonction récursive qui renvoie un triplet (taille minimum d'un ensemble dominant, taille minimum d'un ensemble dominant contenant la racine, taille minimum d'un ensemble dominant ne contenant pas forcément la racine).

IV Recherche de doublon

IV.1 Doublon dans un tableau

Soit t un tableau de taille n dont les éléments sont entre 0 et $n-1$ (inclus).
On veut déterminer si t contient un doublon, c'est-à-dire un élément apparaissant plusieurs fois.

1. Donner un algorithme en complexité temporelle $O(n)$ pour résoudre ce problème. Quelle est la complexité spatiale ?

> Solution : On parcourt le tableau et on utilise un tableau de booléens pour savoir si on a déjà rencontré un élément.

2. Peut-on adapter l'algorithme précédent si les éléments de t ne sont pas entre 0 et $n-1$? pas forcément entiers ?

> Solution : On utilise une table de hachage au lieu d'un tableau de booléens.

3. On reprend l'hypothèse où les éléments de t sont entre 0 et $n-1$.
Décrire un algorithme en complexité $O(n)$ en temps et $O(1)$ en mémoire. On pourra modifier t.

> Solution : On parcourt `t` et on utilise les cases du `t` pour marquer les éléments déjà rencontrés : en voyant `t[i]`, on modifie `t[t[i]] = t[t[i]] + n` de façon à savoir si on a déjà rencontré `t[i]` (il suffit de tester si `t[t[i]] >= n`).

```
bool has_double(int t[], int n) {
    for (int i = 0; i < n; i++) {
        int j = t[i] % n; // pour obtenir la valeur initiale de t[i]
        if (t[j] >= n) {
            return true;
        }
        t[j] += n;
    }
    return false;
}
```

IV.2 Cycle dans une liste chaînée

On considère un type `linked_list` de liste simplement chaînée impérative (chaque élément a accès à l'élément suivant `next`) :

```
typedef struct cell {
    int elem;
    struct cell *next;
} cell;

typedef cell *linked_list;
```

Il est possible qu'une liste chaînée `l` possède un cycle, si l'on revient sur le même élément après avoir parcouru plusieurs successeurs.

4. Décrire un algorithme naïf pour tester si `l` contient un cycle. Quelle est sa complexité en temps et en espace ?

> Solution : On peut utiliser un ensemble (`set`) pour stocker les éléments déjà rencontrés et tester l'appartenance en $O(1)$.

L'algorithme de Floyd est plus efficace. Il consiste à initialiser une variable `tortue` au premier élément de `l`, une variable `lievre` à la case suivante, puis, tant que c'est possible :

- Si `lievre` et `tortue` font référence à la même case, affirmer que `l` contient un cycle.
- Sinon, avancer `lievre` de deux cases et `tortue` d'une case.

5. Montrer que cet algorithme permet bien de détecter un cycle dans `l`. Quelle est l'intérêt de cet algorithme par rapport à celui de la question 4 ?

> Solution : Supposons qu'il existe un cycle de taille p. Alors, au bout d'un moment, `lievre` et `tortue` seront dans le cycle, disons à des positions ℓ et t à partir du début du cycle. Comme `lievre` avance de 1 case relativement à `tortue`,

`lievre` et `tortue` seront sur la même case au bout de $(\ell - t) \mod p$ itérations.
S'il n'y a pas de cycle, il est évident que `lievre` et `tortue` ne seront jamais sur la même case.
Complexité en espace : 2 (stocker `lievre` et `tortue`...). Complexité en temps : $O(n)$.

6. Écrire une fonction `bool has_cycle(linked_list l)` détectant un cycle en utilisant l'algorithme du lièvre et de la tortue.

Solution :

```
bool has_cycle(linked_list l) {
    if (l == NULL) {
        return false;
    }
    linked_list tortue = l;
    linked_list lievre = l->next;
    while (lievre != NULL && lievre != tortue) {
        tortue = tortue->next;
        lievre = lievre->next;
        if (lievre != NULL) {
            lievre = lievre->next;
        }
    }
    return lievre != NULL;
}
```

7. Expliquer comment obtenir la longueur T du cycle ainsi que le nombre d'itérations L avant d'entrer dans le cycle.

Solution : T peut s'obtenir en comptant le nombre d'itérations entre deux rencontres de `lievre` et `tortue`.
Après i itération en partant du début, `lievre` est à la position $2i$ et `tortue` à la position i. `lievre` et `tortue` sont à la même position quand $2(i - L) = i - L \mod T$. On en déduit alors $L = i \mod T$.

Soit t un tableau contenant n entiers entre 0 et $n - 2$ (inclus).

8. Montrer que t contient un doublon.

Solution : Par le principe des tiroirs, il existe $i \neq j$ tels que $t.(i) = t.(j)$.

9. Expliquer comment utiliser l'algorithme de Floyd pour déterminer un doublon de t en complexité $O(n)$ en temps et $O(1)$ en mémoire, sans modifier t.

Solution : On considère la liste chaînée l dont les éléments sont les indices de t, qui commence en $n - 1$ et dont le successeur de i est $t.(i)$. Un doublon correspond au point d'entrée du cycle de l, ce que l'on peut trouver en utilisant la question 7.

IV.3 Presque doublon

On considère un nouveau problème :

Entrée : un tableau t de n entiers et deux entiers a, b.
Sortie : un booléen indiquant s'il existe deux indices $i \neq j$ tels que $|i - j| \leq a$ et $|t[i] - t[j]| \leq b$.

10. Décrire un algorithme en complexité temporelle $O(n)$ en utilisant une table de hachage.

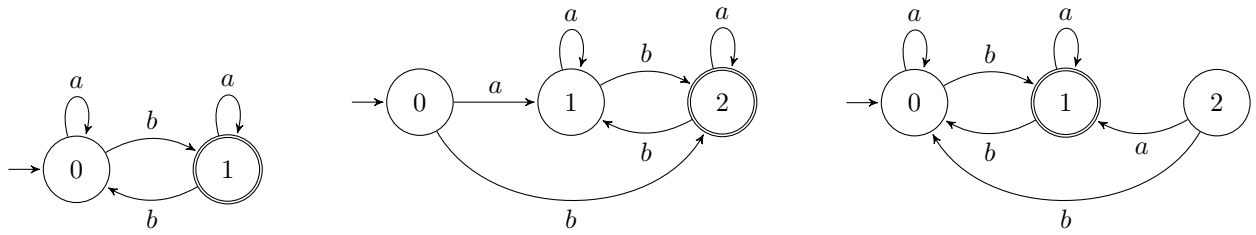
V Morphisme d'automate

Dans toute la suite, les automates sont **déterministes** et **complets**.
On fixe l'alphabet $\Sigma = \{a, b\}$.
Soient deux automates $A = (Q_A, i_A, \delta_A, F_A)$ (où Q_A est l'ensemble d'états, i_A l'état initial, δ_A la fonction de transition et F_A les états finaux) et $A' = (Q_{A'}, i_{A'}, \delta_{A'}, F_{A'})$. Une fonction $f : Q_A \longrightarrow Q_{A'}$ est un **morphisme d'automates** si :

(i) $f(i_A) = i_{A'}$

(ii) $\forall q \in Q_A, \forall a \in \Sigma : \ f(\delta_A(q, a)) = \delta_{A'}(f(q), a)$

(iii) $\forall q \in Q_A : \ f(q) \in F_{A'} \iff q \in F_A$



A_1 A_2 A_3

1. Expliciter un morphisme de A_2 vers A_1.

> Solution : On vérifie que $f(0) = f(1) = 0$ et $f(2) = 1$ convient.

2. Montrer qu'il n'existe pas de morphisme de A_3 vers A_1.

> Solution : Par la propriété (iii), on aurait $f(2) = f(0) = 0$. Mais, par (ii), $f(0) = f(\delta_{A_3}(2, b)) = \delta_{A_1}(f(2), b) = \delta_{A_1}(0, b) = 1$ ce qui est absurde.

3. Montrer que s'il existe un morphisme f de A vers A' alors A et A' acceptent le même langage. La réciproque est-elle vraie ?

> Solution :
> Soit $u \in \Sigma^*$. Comme A et A' sont déterministes complets, $\delta_A^*(i_A, u)$ et $\delta_{A'}^*(i_{A'}, u)$ sont définis.
> On montre alors par récurrence sur $|u|$ que $\boxed{f(\delta_A^*(i_A, u)) = \delta_{A'}^*(i_{A'}, u)}$. D'après (iii), $\delta_A^*(i_A, u)$ est final dans A (donc accepté par A) si et seulement si $f(\delta_A^*(i_A, u))$ est final dans A' (donc accepté par A').

On suppose que A' est **accessible**, c'est-à-dire que tout état de A' est accessible depuis l'état initial i'.

4. On suppose qu'il existe un morphisme f de A vers A'. Montrer que f est surjective.

> Solution : Soit $q' \in Q'$. Comme A' est accessible, il existe $u \in \Sigma^*$ tel que $\delta_{A'}^*(i_{A'}, u) = q'$. On a alors $f(\delta_A^*(i_A, u)) = \delta_{A'}^*(i_{A'}, u) = q'$.

5. Décrire un algorithme permettant de savoir s'il existe un morphisme de A vers A' et de le calculer s'il existe. On précisera les structures de données utilisées et la complexité.

> Solution : On peut calculer le morphisme de proche en proche (par parcours de graphe), en le stockant dans un dictionnaire dont les clés sont les états de A et les valeurs les états de A'.

On définit l'**automate produit** $A \times A' = (Q_A \times Q_{A'}, (i_A, i_{A'}), \delta_{A\times A'}, F_A \times F_{A'})$ où $\delta_{A\times A'}((q, q'), a) = (\delta_A(q, a), \delta_{A'}(q', a))$.

6. On suppose que $A \times A'$ est accessible et $L(A) = L(A')$. Montrer qu'il existe un morphisme de $A \times A'$ vers A (et donc aussi une morphisme de $A \times A'$ vers A').

> Solution : On définit $f : Q \times Q' \longrightarrow Q$ par $f((q, q')) = q$. On vérifie que f est un morphisme de $A \times A'$ vers A.

Soit $B = (Q_B, i_B, \delta_B, F_B)$ un automate accessible. On suppose qu'il existe un morphisme f de B vers A et un morphisme g de B vers A'.
On veut trouver un automate C et des morphismes f' de A vers C et g' de A' vers C.
Pour cela, on introduit la relation d'équivalence suivante sur Q_B :

$p \equiv q \iff \exists q_0, q_1, ..., q_k$ états de Q_B tels que $q_0 = p, q_k = q$ et $\forall i \in [\![0, k-1]\!],\ f(q_i) = f(q_{i+1})$ ou $g(q_i) = g(q_{i+1})$

7. Montrer que si $p \equiv q$ alors $\forall x \in \Sigma,\ \delta_B(p, x) \equiv \delta_B(q, x)$.

> Solution : Supposons $p \equiv q$. Il existe $q_0, q_1, ..., q_k$ états de Q_B tels que $q_0 = p, q_k = q$ et $\forall i \in [\![0, k-1]\!], f(q_i) = f(q_{i+1})$ ou $g(q_i) = g(q_{i+1})$.

8. Montrer que si $p \equiv q$ alors p est un état final si et seulement si q est un état final.

> Solution :

On note $[q]$ la **classe d'équivalence** de $q \in Q_B$ pour la relation $\equiv$ et Q_C l'ensemble des classes d'équivalence de $\equiv$.

9. Décrire un algorithme pour calculer Q_C. On précisera les structures de données utilisées et la complexité.

> Solution : On peut utiliser une structure Union-Find.

10. Définir un automate C dont l'ensemble d'états est Q_C et tel que $h : q \longrightarrow [q]$ soit un morphisme de B vers C.

> Solution : On définit $C = (Q_C, [i_B], \delta_C, F_C)$ où $\delta_C([q], x) = [\delta_B(q, x)]$ et $F_C = \{[q] \in Q_C \mid q \in F_B\}$.

11. Définir deux morphismes f' de A vers C et g' de A' vers C tels que $f' \circ f = h = g' \circ g$.

> Solution : Soit $q_A \in Q_A$. Comme f est surjective, il existe $q_B \in Q_B$ tel que $f(q_B) = q_A$. On pose alors $f'(q_A) = [q_B]$.

Soit L un langage rationnel et n le plus petit nombre d'états d'un automate reconnaissant L.

12. Montrer que deux automates à n états reconnaissant L sont isomorphes, c'est à dire qu'il existe un morphisme bijectif de l'un à l'autre.