# TP-Colle-4 (MPI*) : Automates et graphes

## Environnement

Nous travaillerons en OCaml mode interprété ET en C pour ce TP-colle. Nous vous laissons le choix de l'environnement, selon vos habitudes. Nous recommandons toutefois :

- L'interpréteur en ligne BetterOCaml : https://BetterOCaml.ml/
- Le compilateur en ligne OnlineGDB : hhttps://www.onlinegdb.com/

## Consignes et mise en place

- Vous répondrez aux questions de programmation en écrivant le code dans votre outil.
- Vous répondrez aux questions de cours à l'aide de la fiche-réponse à la fin du sujet.
- Vous répondrez aux questions à développer à l'oral lors de votre passage au tableau (pour les concernés). Utilisez un brouillon pour vous préparer des notes.

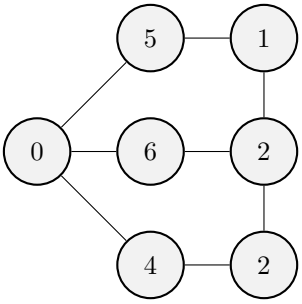
## I Préliminaires

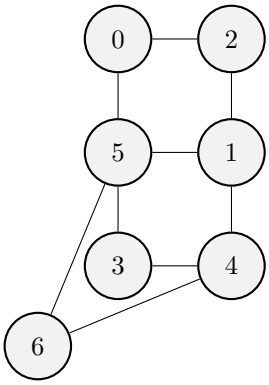
1. *(Question de cours)* Donnez le pseudo-code de l'algorithme de Kruskal.
2. Si vous ne travaillez pas sur la section V, récupérez le fichier `graphs.c` (sur Cahier-de-Prépa ou sur le réseau) et complétez dans le code fourni :
   - la fonction `create_graph`, qui fait les allocations dynamique nécessaires pour compléter les champs du pointeur `g` donnés en entrée. Les arêtes dans la matrice d'adjacence seront toutes initialités à `false` ; pour faire des graphes d'exemple, vous intialiserez manuellement leurs arêtes à `true` dans le main.
   - la fonction `release_graph`, qui libère la mémoire utilisée par le pointeur donné en entrée.

## II Graphe biparti et bicoloration (en C)

Dans cette section, on représentera la couleur rouge par l'entier 0, et la couleur verte par l'entier 1. Un graphe non-orienté $G = (S, A)$ est *bicolorable* s'il existe une coloration $c : S \to \{0, 1\}$ des sommets du graphe telle que, pour toute arête $(u, v) \in A$, $c(u) \neq c(v)$.

1. Modifiez la fonction `set_edge` pour qu'elle crée des arêtes en traitant le graphe donné en argument comme un graphe non-orienté.
2. Créez trois constantes de type `int` pour représenté les cas `VERT`, `ROUGE`, et `NON_COLORE`.
3. Modifiez votre type `struct graph` pour y ajouter un tableau représentant la couleur de chaque sommet ; modifiez aussi la fonction `create_graph` afin d'initialiser chaque sommet à `NON_COLORE`.
4. *(Question à développer à l'oral)* Les deux graphes suivants admettent-ils sont-ils bicolorable ?




5. Implémentez ces deux graphes avec les types et fonctions fournies.
6. *(Question à développer à l'oral)* Montrer qu'un graphe non-orienté est biparti si et seulement s'il est bicolorable.
7. *(Question à développer à l'oral)* On propose l'algorithme suivant pour montrer qu'un graphe non-orienté est biparti en tentant de construire une bicoloration, dans le cas d'un graphe connexe.

   (i) Créer un tableau `visite`, pour indiquer quels sommets ont été vus.
   (ii) Colorier le sommet 0 en rouge.
   (iii) Tant qu'il existe un sommet $i$ coloré et non-examiné, faire :
   - indiquer que le sommet $i$ est visité.
   - si $i$ est rouge (resp. vert), alors :
     – Pour chaque voisin $j$ de $i$, si $j$ est rouge (resp. vert), on s'arrête et on renvoie faux. Sinon, on colorie $j$ en vert (resp. rouge) s'il est non-coloré.

   (iv) Si l'algorithme ne s'est pas arrêté avant, on renvoie vrai.

   En vous appuyant sur un invariant bien choisi, montrer que cet algorithme est correct.
8. Implémentez une fonction `bool est_biparti(graph* g)` qui réalise l'algorithme précédent. Quelle est sa complexité ?
9. Comment proposez-vous de modifier l'algorithme précédent pour l'appliquer à des graphes non-connexes ? Implémentez votre modification.On souhaite maintenant étendre le problème de la coloration : étant donné

   un graphe non-orienté $G = (S, A)$ arbitraire, on souhaite construire une coloration de $G$, c'est-à-dire une fonction $c : S \to [\![0; p-1]\!]$ de sorte qu'aucun sommet n'ait la même couleur que son voisin.

10. Implémentez une fonction `void colore(graph* g)` qui réalise une coloration d'un graphe.[1]

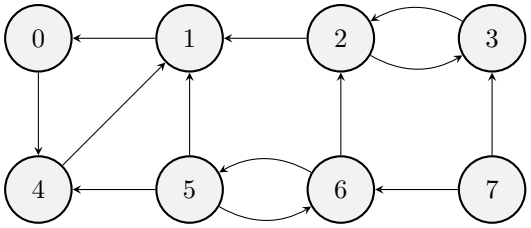
# III Composantes fortement connexes : méthode matricielle (en C)

Etant donné un graphe orienté $G = (S, A)$ avec $S$ de la forme $[\![0, n-1]\!]$, on définit formellement l'ensemble $\{Vrai, Faux\}$, et la *matrice d'adjacence* $B = (b_{ij})_{i,j \in S}$ comme une matrice de booléens : $b_{ij}$ vaut $Vrai$ si $(i, j) \in A$ et $Faux$ sinon.

On ne considérera que des matrices carrées ici. Dans l'implémentation, une matrice carrée booléenne de taille $n \times n$ sera considérée comme un tableau de type `bool` à une dimension, à `n` × `n` cases.

1. Utilisez les fonctions fournies pour implémenter le graphe d'exemple suivant :

[1] Ne cherchez pas à le construire nécessairement avec le nombre minimum de couleur possible : déterminer ce nombre est un problème complexe algorithmiquement.




2. *(Question à développer à l'oral)* Proposez une fonction $f : [\![0, n-1]\!] \times [\![0, n-1]\!] \to [\![0, n^2-1]\!]$, qui à la case $(i, j)$ d'une matrice booléenne associe l'indice de la case dans le tableau de `bool` correspondant.
3. On définit la somme et le produit de matrices booléennes comme pour des matrices sur un corps $K$, où les opérateurs booléens $\vee$ et $\wedge$ jouent respectivement le rôle de l'addition et de la multiplication.

   Implémentez deux fonctions `void somme(int n, bool* m1, bool* m2, bool* m3)` et `void produit(int n, bool* m1, bool* m2, bool* m3)`, qui prennent deux matrices `m1` et `m2` de taille `n` $\times$ `n`, et écrit dans la matrice `m3` le résultat de leur somme et de leur produit.
4. Ecrivez une fonction `void puissance(int n, bool* m1, int p, bool* m2)` qui prend une matrice `m1` de taille `n` $\times$ `n`, l'élève à la puissance `p` et stocke le résultat dans `m2`.
5. *(Question à développer à l'oral)* Si $B$ est la matrice d'adjacence d'un graphe orienté $G$, montrer que :

   $$B^k = (m_{ij})_{i,j \in S} \text{ où "}m_{ij} \text{ est vrai si et seulement s'il existe un chemin de } k \text{ arêtes de } i \text{ à } j\text{".}$$

6. *(Question à développer à l'oral)* On pose $B_k = \sum_{i=0}^{k} B^i$. Montrer que :

   $$B_k = (p_{ij})_{i,j \in S} \text{ où "}p_{ij} \text{ est vrai si et seulement s'il existe un chemin d'au plus } k \text{ arêtes de } i \text{ à } j\text{."}$$

   Montrez que la suite $(B_k)$ converge vers une matrice $B_*$. Que représente $B_*$ ?
7. Ecrivez une fonction `void chemins(graph* g, bool* m)` qui calcule la matrice précédente pour le graphe `g` et stocke le résultat dans `m`.
8. *(Question à développer à l'oral)* Proposez une méthode pour calculer les composantes fortement connexes du graphe à partir de la matrice $B_*$.
9. Implémentez la méthode précédente dans une fonction `void cfc (graph* g, unsigned int* tab)`. Cette fonction stockera dans le tableau `tab` de taille $|S|$ dans chaque case $i$ un représentant de la composante fortement connexe de $i$.
10. *(Question à développer à l'oral)* Quelle est la complexité totale du calcul des composantes fortement connexes à partir de la donnée du graphe, avec la méthode employée ?

# IV Graphes 2-connexes (en C)

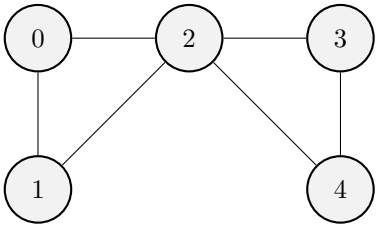
Un graphe $G = (S, A)$ non-orienté et connexe est dit 2-*connexe* si, pour tout $s \in S$, le graphe $G$ auquel on enlève le sommet $s$ (ainsi que ses arêtes) reste connexe.

Chercher si un graphe est 2-connexe est un problème proposant de multiples applications. Par exemple, en visualisant un réseau de communication interne à une entreprise comme un graphe, où des câbles relient différents routeurs, si le réseau est 2-connexe, alors un des routeurs peut tomber en panne sans que cela ne coupe le réseau en deux parties séparées. On a alors "le droit" à une panne sans pénaliser le réseau, qu'on va alors tenter de réparer rapidement.

La notion de 2-connexité se reformule en terme de *point d'articulation* : un point d'articulation du graphe connexe $G$ est un sommet $s \in S$ tel que le graphe $(S \backslash \{s\}, A \backslash \{(s, s') \mid s' \in S\})$ n'est pas connexe. Alors un graphe 2-connexe est un graphe qui n'admet aucun point d'articulation.

1. Modifiez la fonction `set_edge` pour qu'elle crée des arêtes en traitant le graphe donné en argument comme un graphe non-orienté.

2. *(Question à développer à l'oral)* Le graphe-ci-dessous est-il 2 connexe ? Comment peut-on le modifier pour qu'il le devienne ?




3. *(Question à développer à l'oral)* Rappelez comment déterminer algorithmiquement les composantes connexes d'un graphe non-orienté.
4. *(Question à développer à l'oral)* Proposez un algorithme naïf basé sur la question précédente pour déterminer si un graphe est 2-connexe. Evaluez sa complexité.
5. Implémentez une fonction `void visiter(const graph g, int s, int* visite)`, qui réalise la visite du sommet `t` lors d'un parcours en profondeur, en recevant et modifiant le tableau `visite` des sommets déjà visités.
6. Implémentez une fonction `int nb_composantes_connexes(const graph g)` qui calcule le nombre de composantes fortement connexes d'un graphe donné.
7. Implémentez une fonction `bool est_2_connexe(const graph g)` qui utilise la méthode naïve donnée précédemment pour déterminer si un graphe est 2-connexe. Testez-la sur le graphe d'exemple ci-dessus, ainsi que sur sa version modifiée.
8. *(Question à développer à l'oral)* Donnez la complexité de l'algorithme précédent.[2]

# V Algorithme de Moore (OCaml)

Soit $A = (Q, \Sigma, \{q_0\}, F, \delta)$ un automate déterministe et complet. On considérera $\delta$ comme une fonction de $Q \times \Sigma$ dans $Q$, qu'on se permettra d'étendre en une fonction de $Q \times \Sigma^*$ dans $Q$. Ainsi, $\delta(q, a)$ (resp. $\delta(q, u)$) désigne l'unique état où l'on arrive en lisant la lettre $a$ (resp. le mot $u$) depuis l'état $q$, s'il existe.
On cherche à construire un automate déterministe reconnaissant le même langage que $\mathcal{A}$, mais ayant un nombre minimal d'états. On appellera *automate minimal* un tel état.

Plusieurs algorithmes existent pour construire un tel état. On va s'intéresser ici à l'algorithme de Moore. On introduit les notations suivantes :

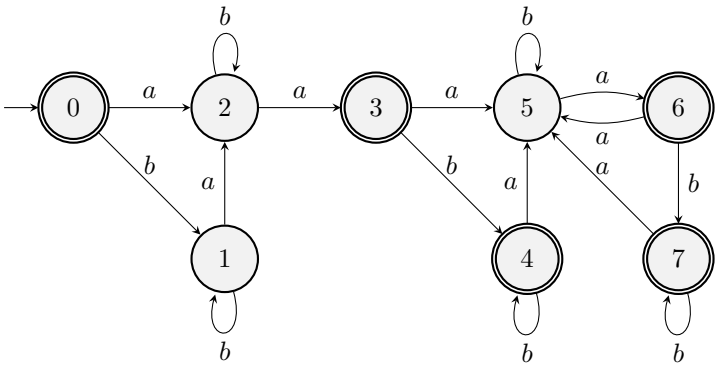
- Pour $q \in Q$, $h \in \mathbb{N}$, on note $L_q^{(h)}(\mathcal{A}) = \{u \in \Sigma^* \mid \delta(q, u) \in F, |u| \leq h\}$ le langage des mots de longueur au plus $h$ qui permettent d'atteindre un état acceptant depuis $q$.
- On introduit une suite de relations d'équivalences $\sim_0, \sim_1$, etc... dans $Q$, de sorte que :

$$p \sim_h q \Leftrightarrow L_p^{(h)}(\mathcal{A}) = L_q^{(h)}(\mathcal{A})$$

  Ainsi, $p \sim_h q$ signifie que ce sont exactement les mêmes mots de longueur au plus $h$ qui permettent d'atteindre un état acceptant. On peut voir cette relation sous la forme : "$p$ et $q$ jouent le même rôle du point de vue de qu'il reste à lire".

1. Définissez un type `'a automate` représentant un automate déterministe et complet. Vous veillerez à y inclure un tableau `sigma` représentant l'alphabet. Le choix vous est laissé pour l'implémentation des transitions (liste de triplets, matrice de transition indexée par un couple état/lettre, table de hachage, etc...)
2. Implémentez l'automate donné ci-dessous :

[2] Un algorithme présenté par Hopcroft et Tarjan en 1973 permet de déterminer l'ensemble des points d'articulations d'un graphe en temps $O(|S| + |A|)$, en se basant sur un seul parcours en profondeur. Sa description étant relativement complexe, on se limite ici à une approche naïve.




3. *(Question à développer à l'oral)* Que signifie la relation d'équivalence $\sim_0$ ? Quelles sont ses classes d'équivalence ?
4. *(Question à développer à l'oral)* Montrez que pour tous $p, q \in Q$, $h \in \mathbb{N}$, on a cette équivalence :

$$p \sim_{h+1} q \Leftrightarrow \begin{cases} p \sim_h q \\ \delta(p,a) \sim_h \delta(q,a) \text{ pour toute lettre } a \in \Sigma \end{cases}$$

5. *(Question à développer à l'oral)* Vérifier que si $p \not\sim_h q$, alors pour tout $k \geq h$, on a $p \not\sim_k q$. En déduire la convergence de la suite de relations d'équivalences $(\sim_h)_{h\in\mathbb{N}}$.

   Il se dégage alors un algorithme assez naturel de cette suite de relation :

   - On calcule la suite des relations $(\sim_h)_{h\in\mathbb{N}}$ jusqu'à convergence. Chaque relation donne une partition de $Q$ (l'ensemble de ses classes d'équivalences), qui va s'affiner à chaque étape jusqu'à ce que la conergence soit atteinte.
   - Lorsqu'on a atteint la limite de la suite, on crée un nouvel automate comportant un état par classe d'équivalence. On crée ensuite de manière très intuitive ses états initiaux, acceptants, et ses transitions.

6. *(Question à développer à l'oral)* Appliquer la méthode précédente à l'automate donné en exemple ci-dessus. Construire l'automate obtenu après convergence.

   Pour effectuer le calcul des classes $(\sim_h)_{h\in\mathbb{N}}$ dans l'implémentation, on se propose la méthode suivante :

   - à l'étape $h$, on dispose d'un tableau `r` de taille $|Q|$ tel que pour tout `i`, `r.(i)` est un représentant de la classe d'équivalence de l'état `i`.
   - on crée une matrice `m`, de type `int array array`, telle que pour tout état `i`, et pour la `j`-ème lettre de l'alphabet $\Sigma = \{a_0, a_1, \ldots, a_{l-1}\}$, `m.(i).(j)` contienne un représentant de la classe d'équivalence de $\delta(i, a_j)$ ; on peut chercher ce représentant dans `r`.
   - puis on crée le tableau `r'` qui correspond aux classes d'équivalences de $(\sim_{h+1})$. S'il est égal à `r`, on a convergé. Sinon, on continue.

7. Créer une fonction `relation_suivante : 'a automate -> int array -> int array`, qui prend en entrée un automate déterministe complet et le tableau `r` avec les notations précédentes, et qui renvoie le nouveau tableau `r'`.
8. Créer une fonction `convergence : 'a automate -> int array` qui renvoie le tableau des classes d'équivalence de la limite de la suite de relations pour un automate déterministe complet donné en entrée.
9. Créer une fonction `renommage : int array -> int array` qui reçoit un tableau `r` de classes d'équivalence produit par convergence, et produit un nouveau tableau `s` qui conserve ces classes d'équivalence (i.e. `r.(i) = r.(j)` ssi `s.(i) = s.(j)`), en les renumérotant par les entiers d'un intervalle $[\![0; p-1]\!]$, où $p$ est le plus petit possible.
10. Créer une fonction `moore : 'a automate -> 'a automate` qui construit l'automate minimal dont les états sont les classes d'équivalence renommée par la fonction précédente.
11. *(Question à développer à l'oral)* Quelle est la complexité de toute cette transformation ?

# Fiche réponse

NOM :
Prénom :
Répondez ici aux questions de cours, en rédigeant vos réponses.