# **DS2** (sujet type CCINP)

## 1 Problème sur les arbres couvrants (C)

Dans ce problème, un graphe $G$ est un couple $(S, A)$ où :

- $S$ est un ensemble fini dont les éléments sont les sommets de $G$ ;
- $A = (a_0, a_1, \ldots, a_{m-1})$ est la suite des arêtes de $G$, une arête étant une partie $a = \{s, t\}$ de $S$ de cardinal 2. Les sommets $s$ et $t$ sont appelés les extrémités de l'arête $a$ et on dira que $a$ relie $s$ et $t$. Si $s$ et $t$ sont reliés par une arête, on dit qu'ils sont voisins ou adjacents.

Ainsi, les graphes sont non orientés et il n'y a pas d'arête reliant un sommet à lui-même.
Par convention, nous noterons $n$ (respectivement $m$) le nombre de sommets (respectivement d'arêtes) du graphe et nous supposerons que $S = \{0, 1, \ldots, n-1\} = S_n$.

Un graphe sera représenté par le type C suivant :

```
struct graphe = {int size;
                 int* degrees;
                 int** adj;};

typedef struct graphe graphe;
```

Si $G$ est un graphe représenté par `g`, alors le nombre de sommets `n` du graphe est donné par le champs size, le tableau `g.degrees` de taille $n$ contient dans sa case d'indice `i` le degré du sommet `i` et le tableau `g.adj[i]` est de taille `g.degrees[i]` et contient les voisins du sommet `i` dans $G$.

Soit $G = (S, A)$ un graphe.

- Un chemin dans $G$ est une suite $c = (s_0, s_1, \ldots, s_{j-1}, s_j, \ldots, s_{k-1}, s_k)$ où pour tout $j$ compris entre 1 et $k$, $s_{j-1}$ et $s_j$ sont des sommets voisins. On dira que $c$ est un chemin de $s_0$ à $s_k$ de longueur $k$. Par convention, pour $s$ sommet de $G$, il existe un chemin de longueur nulle de $s$ à $s$.
- La composante connexe d'un sommet $s$ de $G$, notée $C_s$ , est l'ensemble des sommets $t$ de $G$ tels qu'il existe un chemin de $s$ à $t$.
- On dit que $G$ est connexe si pour tous sommets $s$ et $t$ de $G$, il existe un chemin de $s$ à $t$.
- Un cycle dans $G$ est un chemin de longueur $k \geq 2$ d'un sommet à lui-même et dont les arêtes sont deux à deux distinctes. On dit que $G$ est acyclique s'il ne contient aucun cycle.
- Un arbre est un graphe connexe acyclique.

### 1.1 Caractérisation des arbres

Soit $G = (S_n, A)$ un graphe à $n$ sommets et $m$ arêtes, représenté par `g`. Les arêtes de $G$ sont donc numérotées $A = (a_0, a_1, \ldots, a_{m-1})$.

**Q1** Montrer que les composantes connexes de $G$ forment une partition de $S_n$ , c'est-à-dire que :

1. $\forall s \in S_n, C_s \neq \emptyset$ ;
2. $S_n = \bigcup_{s \in S_n} C_s$ ;
3. Pour tous sommets $s$ et $t$, soit $C_s = C_t$ , soit $C_s \cap C_t = \emptyset$.

**Q2** Montrer que si $s$ et $t$ sont deux sommets de $G$ tels que $t \in C_s$, il existe un plus court chemin de $s$ à $t$ et que les sommets d'un plus court chemin sont deux à deux distincts.

Pour $k \in \{0, 1, \ldots, m\}$, $G_k$ désigne le graphe $(S_n, (a_0, a_1, \ldots, a_{k-1}))$ obtenu en ne conservant que les $k$ premières arêtes de $G$.

**Q3** On suppose que $G$ est un arbre. Montrer que pour tout $k \in \{0, 1, \ldots, m-1\}$, les extrémités de $a_k$ appartiennent à deux composantes connexes différentes de $G_k$. En déduire que $m = n - 1$.

**Q4** Montrer que les trois propriétés suivantes sont équivalentes :

1. $G$ est un arbre ;
2. $G$ est connexe et $m = n - 1$.
3. $G$ est acyclique et $m = n - 1$.

Nous souhaitons écrire une fonction qui teste si un graphe est un arbre. Nous allons pour cela utiliser une structure de données permettant de manipuler les partitions de l'ensemble $S_n$ : si $\mathcal{P} = \{X_1, X_2, \ldots, X_k\}$ est une partition de $S_n$, on choisit dans chaque partie $X_i$, un élément particulier $r_i$, appelé représentant de $X_i$. Notre structure de données doit nous permettre :

- de calculer, pour $s \in S_n$ , le représentant de la partie $X_i$ contenant $s$ ; cet élément sera également appelé représentant de $s$ ;
- pour deux entiers $s$ et $t$ représentant des parties distinctes $X_i$ et $X_j$, de transformer $\mathcal{P}$ en réunissant $X_i$ et $X_j$, $s$ ou $t$ devenant le représentant de la partie $X_i \cup X_j$.

Nous représenterons une partition $\mathcal{P} = \{X_1, X_2, \ldots, X_k\}$ de $S_n$ par une forêt : chaque $X_i$ est représenté par un arbre dont les noeuds sont étiquetés par les éléments de $X_i$ et de racine le représentant $r_i$ de $X_i$ , les arcs étant orientés vers la racine. Nous noterons $h(r_i)$ la hauteur de l'arbre $X_i$ , c'est-à-dire la longueur de sa plus longue branche.

Ainsi, $\mathcal{P}_9 =$ `{{0, 2}, {1, 5, 6, 8}, {3, 4, 7}}` est une partition de $S_9$ et peut par exemple être représentée par la forêt de la figure suivante :

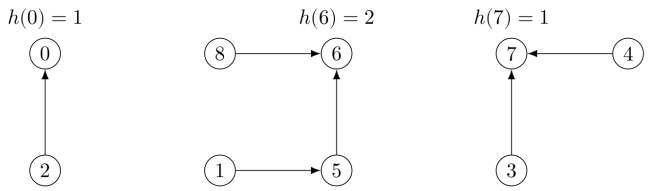


**Figure 3** Une représentation de $\mathcal{P}_9 = \{\{0, 2\}, \{1, 5, 6, 8\}, \{3, 4, 7\}\}$

Le calcul du représentant d'un entier $s \in S_n$ se fait donc en remontant jusqu'à la racine de l'arbre (ce qui justifie l'orientation des arcs). Dans l'exemple, les représentants de 2, 1 et 7 sont respectivement 0, 6 et 7. Une partition $\mathcal{P}$ de $S_n = \{0, 1, \ldots, n-1\}$ sera représentée par un tableau d'entiers `P` de taille `n` de sorte que pour tout $s \in \{0, 1, \ldots, n-1\}, P[s]$ est le père de $s$ si $s$ n'est pas un représentant, et est égal à $-1 - h(s)$ si $s$ est un représentant. La partition $\mathcal{P}_9$ peut ainsi être représentée par le tableau : `{-2, 5, 0, 7, 7, 6, -3, -2, 6}`

La réunion de deux parties $X_i$ et $X_j$ de représentants $s$ et $t$ distincts se fait selon la méthode suivante :

- si $h(s) > h(t)$, $s$ est choisi pour représentant de la partie $X_i \cup X_j$ et devient le père de $t$ ;
- si $h(s) \leq h(t)$, $t$ est choisi pour représentant de la partie $X_i \cup X_j$ et devient le père de $s$.

**Q5** Donner le résultat de la réunion de $X_1$ et $X_2$ dans $\mathcal{P}_9$.

**Q6** Écrire une fonction `int representant(int* tab, int size, int s)` qui, appliquée à un tableau représentant une partition de $S_n$ de taille `size` et à $s \in S_n$ , renvoie le représentant de $s$. On demande que la fonction proposée soit récursive.

**Q7** Écrire une fonction `void union(int* tab, int size, int s, int t)` qui, appliquée à un tableau représentant une partition de $S_n$ de taille `size` et à deux représentants $s$ et $t$ distincts, modifie la partition en réunissant les arbres associés à $s$ et $t$, selon la méthode expliquée ci-dessus, sans oublier, si nécessaire, de modifier $h(s)$ ou $h(t)$.

On note $\mathcal{P}_n(0)$ la partition de $S_n$ où toutes les parties sont des singletons.

**Q8** Soit $\mathcal{P}$ une partition de $S_n$ construite à partir de $\mathcal{P}_n(0)$ par des réunions successives selon la méthode précédente. Montrer que si $s$ est le représentant d'une partie $X \in \mathcal{P}$, alors le cardinal de $X$ vérifie $|X| \geq 2^{h(s)}$.

**Q9** En déduire les complexités des deux fonctions précédentes dans le pire des cas en fonction de $n$ pour une partition $\mathcal{P}$ construite à partir de $\mathcal{P}_n(0)$.

**Q10** Citer et expliquer le principe de la méthode au programme qui permet d'améliorer la complexité amortie de ces opérations. Ecrire une fonction `representant_opt` qui inclut cette optimisation.

**Q11** Écrire une fonction `bool est_un_arbre(graphe g)` qui, appliquée à un graphe `g` représentant un graphe $G$, renvoie `true` si $G$ est un arbre et `false` sinon.

## 1.2 Algorithme de Wilson : arbre couvrant aléatoire

Soit $G = (S_n, A)$ un graphe connexe et soit un entier $r \in S_n$. Un arbre couvrant de $G$ enraciné en $r$ est un arbre $T = (S_n, B)$ tel que $B \subset A$ et dont $r$ a été choisi pour racine.

On convient de représenter un tel arbre en suivant la même idée que pour représenter une partition dans la partie précédente : par un tableau tel que la case $s$ du tableau contient l'indice du père de $s$ dans l'arbre si $s$ n'est pas lui-même la racine, et $-1$ sinon :

```
struct arbre = {int size; int* peres;};
typedef struct arbre arbre;
```

Dans cette partie, nous supposons que $G = (S_n, A)$ est un graphe connexe et $r$ un sommet de ce graphe. Nous cherchons à construire un arbre couvrant aléatoire de $G$ enraciné en $r$.

Nous allons pour cela faire évoluer dynamiquement un arbre $T$ , représenté par un tableau parent de longueur $n$ vérifiant :

- la case $r$ de parent contient $-1$ ;
- si $s \in S_n$ n'est pas un sommet de $T$, alors la case $s$ de parent contient $-2$;
- si $s \in S_n$ est un sommet de $T$ différent de la racine, alors la case $s$ de parent contient le père de $s$ dans $T$.

La construction de l'arbre $T$ se fait en suivant l'algorithme 1.

**Algorithme 1** Algorithme de Wilson de création d'un arbre couvrant aléatoire

**entrée :** $G$ graphe connexe, $r$ sommet de $G$
**sortie :** arbre couvrant aléatoire de $G$ enraciné en $r$

```
1 : T ← ({r}, ∅)
2 : pour tout sommet s de G faire
3 :     si s n'est pas un sommet de T alors
4 :         c ← chemin-aléatoire(G, T, s)
5 :         greffer(c, T)
6 :     fin si
7 : fin pour
8 : renvoyer T
```

La greffe d'un chemin élémentaire qui termine par un sommet de $T$ et dont c'est le seul sommet en commun avec $T$ se fait de manière naturelle, en ajoutant ce chemin.

Le calcul d'un chemin aléatoire dans le graphe entre un sommet qui n'appartient pas à l'arbre et un sommet qui appartient à l'arbre se fait en suivant l'algorithme 2.

**Algorithme 2** Algorithme chemin-aléatoire

**entrée :** $G$ graphe connexe, $T$ arbre sous-graphe de $G$, $s$ sommet de $G$ et pas de $T$
**sortie :** $c = (s_0 = s, s_1, \ldots, s_k)$ chemin aléatoire élémentaire de $G$ partant de $s$ dont le seul sommet dans $T$ est $s_k$

$c \leftarrow (s)$
**tant que** le dernier sommet de $c$ n'appartient pas à $T$ **faire**
    on note $c = (s_0, s_1, \ldots, s_k)$
    soit $u$ un voisin de $s_k$ dans $G$ (choisi aléatoirement et uniformément)
    **si** il existe $i \in [\![0, k]\!]$ tel que $u = s_i$ **alors**
        $c \leftarrow (s_0, \ldots, s_i)$
    **sinon** $c \leftarrow (s_0, \ldots, s_k, u)$
    **fin si**
**fin tant que**
**renvoyer** $c$

On représente un chemin élémentaire en C par le type :

```
struct chemin = {int debut; int fin; int* suivant; int size;};
typedef struct chemin chemin;
```

de telle sorte que si le chemin $c = (s_0, s_1, \ldots, s_k)$ est représenté par $c$, alors :

- le champ `debut` de `c` contient $s_0$,
- son champ `fin` contient $s_k$,
- son champ `suivant` est un tableau de taille `size` et pour tout $j \in \{0, \ldots, k-1\}$, la case d'indice $s_j$ de ce tableau contient la valeur $s_{j+1}$,
- les valeurs des cases du champ `suivant` dont les indices n'appartiennent pas à $\{s_0, \ldots, s_{k-1}\}$ n'ont pas d'importance.

**Q12** Expliquer pourquoi le type `chemin` ne peut pas représenter un chemin non élémentaire.

**Q13** Écrire une fonction `void greffer(arbre* a, chemin* c)` qui implémente la greffe d'un chemin sur un arbre, en supposant (sans avoir à le vérifier) que le chemin possède un unique sommet qui appartient à l'arbre : son dernier sommet.

**Q14** Écrire une fonction `chemin* chemin_aleatoire(graphe g,arbre* a,int s)` qui exécute l'algorithme 2.
On pourra utiliser la commande `rand()%n` quirenvoie uniformément un entier compris entre 0 et $n-1$.

**Q15** Écrire une fonction `arbre* wilson(graphe g,int r)` qui implémente l'algorithme 1. On prendra les initiatives nécessaires pour éviter les fuites mémoire tout en respectant les prototypes des fonctions imposées.

# 2 Résiduels d'un langage

Soit $\Sigma$ un alphabet fini non vide. Soit $L \subset \Sigma^*$, et soit $u \in \Sigma^*$, on pose :

$$u^{-1}L = \{v \in \Sigma^* : uv \in L\}$$

Un tel ensemble est appelé résiduel de $L$. On souhaite montrer qu'un langage est reconnaissable ssi il possède un nombre fini de résiduels.

**Q16** Soient $u, v \in \Sigma^*$ et $L \subset \Sigma^*$, établir une relation entre $(uv)^{-1}L$ et $v^{-1}(u^{-1}L)$. On justifiera cette relation.

**Q17** On fixe ici $\Sigma = \{a, b\}$ et on considère $L$ représenté par l'expression régulière $a(ba + a)^*$.
Représenter graphiquement un automate qui reconnait $L$ puis donner les différents résiduels du langage $L$.

**Q18** On revient au cas général et on fixe $L \subset \Sigma^*$.
On suppose que l'ensemble des résiduels de $L$, $R = \{u^{-1}L : u \in \Sigma^*\}$ est fini.
On pose $\mathcal{A} = (R, i_0, F, \delta)$ un automate déterministe tel que $\forall u^{-1}L \in R, \forall \alpha \in \Sigma, \delta(u^{-1}L, \alpha) = (u\alpha)^{-1}L$.

1. Cette construction n'est pas forcément bien définie, justifier qu'elle l'est. Proposer une définition plus propre qui ne poserait pas ce questionnement.
2. Déterminer une formule pour la fonction étendue de $\delta$ et la prouver.
3. Déterminer $i_0$ et $F$ pour que le langage reconnu par $\mathcal{A}$ soit bien $L$. Prouver que votre proposition convient.
4. Représenter l'automate obtenu pour le langage de la question précédente.

**Q19** Réciproquement, on se donne un langage $L$ reconnaissable et $\mathcal{A} = (Q, q_0, F, \delta)$ un automate déterministe complet le reconnaissant. Pour $q \in Q$, on pose $L_q$ le langage reconnu par $(Q, q, F, \delta)$.

1. Soit $u \in \Sigma^*$, établir un lien entre $u^{-1}L$ et $L_{\delta^*(q_0,u)}$.
2. Démontrer que $L$ n'a qu'un nombre fini de résiduels.

# 3 Traversée de rivière (Ocaml)

Dans une vallée des Alpes, un passage à gué fait de cailloux permet de traverser la rivière. Deux groupes de randonneurs arrivent simultanément sur les berges gauche et droite de cette rivière et veulent la traverser. Le chemin étant très étroit, une seule personne peut se trouver sur chaque caillou de ce chemin (**figure 1**). Un randonneur sur la berge de gauche peut avancer d'un caillou (vers la droite sur la **figure 1**) et sauter par dessus le randonneur devant lui (un caillou à droite) si le caillou où il atterit est libre. De même, chaque randonneur de la berge de droite peut avancer d'un caillou (vers la gauche sur la **figure 1**) et sauter par dessus le randonneur devant lui, dans la mesure où le caillou sur lequel il atterit est libre. Une fois engagés, les randonneurs ne peuvent pas faire marche arrière. De plus, pour simplifier, on suppose qu'une fois tous les randonneurs sur le chemin, il ne reste qu'un caillou de libre.

Figure 1: Les randonneurs et le chemin de cailloux.

Le chemin de cailloux est défini par un tableau d'entiers :

```
type chemin_caillou = int array
```

Dans ce tableau, un randonneur venant de la berge de gauche est représenté par un 1, un randonneur issu de la berge de droite par un 2, et un caillou libre par un 0.

**Q20** Écrire une fonction de signature `caillou_vide : chemin_caillou -> int` qui détermine la position du caillou inoccupé.

**Q21** Écrire une fonction de signature `echange : chemin_caillou -> int -> int -> chemin_caillou` qui permute les valeurs codées sur deux cailloux. Le tableau d'entiers initial représentant le chemin n'est pas modifié. On pourra utiliser ici la fonction `copy` du module `Array`.

**Q22** Écrire une fonction de signature `randonneurG_avance : chemin_caillou -> bool` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse avancer (vers la droite).

**Q23** Écrire une fonction de signature `randonneurG_saute : chemin_caillou -> bool` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse sauter (vers la droite) au-dessus d'un randonneur.

On supposera dans la suite les fonctions de signature
`randonneurD_avance : chemin_caillou -> bool` et `randonneurD_saute : chemin_caillou -> bool` écrites de manière similaire pour les randonneurs venant de la berge de droite.

**Q24** Écrire une fonction de signature `mouvement_chemin : chemin_caillou -> chemin_caillou list` qui, en fonction de létat du chemin, calcule l'état suivant après les opérations suivantes (si elles sont permises) :

(i) déplacement d'un randonneur venant de la berge de gauche,

(ii) déplacement d'un randonneur venant de la berge de droite,

(iii) saut d'un randonneur venant de la berge de gauche,

(iv) saut d'un randonneur venant de la berge de droite.

On donne la syntaxe OCaml pour crréer une liste de $N$ entiers $i$ : `List.init N (fun x -> i)`.
Par exemple, `List.init 5 (fun x -> 2);;` renvoie `[2;2;2;2;2]`.

**Q25** Écrire une fonction de signature `passage : int -> int -> chemin_caillou list`, utilisant la question précédente, telle que l'appel `passage nG nD` résout le problème du passage de `nG` randonneurs venant de la berge de gauche et `nD` randonneurs venant de la berge de droite. Par exemple, `passage 3 2` permet de passer de `[1;1;1;0;2;2]` à `[2;2;0;1;1;1]`.

# 4 Le jeu de Marienbad

Soit $k \in \mathbb{N}^*$. Sur une table sont disposés $k$ tas d'allumettes : le premier tas contient $N_1$ allumettes, le second $N_2$, et le dernier tas contient $N_k$ allumettes. On suppose les $N_i$ strictement positifs. Le jeu se joue à deux. Le premier joueur retire un nombre strictement positif d'allumettes d'un tas, le second joueur fait de même, puis le premier, etc. La partie s'arrête lorsque toutes les allumettes ont été retirées; le joueur gagnant étant celui qui a pris la ou les dernières allumettes présentes sur la table, l'autre étant déclaré perdant.

La table du vérité du «ou exclusif», noté $\oplus$ est :

| $\oplus$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

On rappelle que $\oplus$ définit une structure de groupe commutatif sur $\{0,1\}$. Soient $n$ et $m$ des entiers naturels. Leurs décompositions en base 2 s'écrit :

$$n = \sum_{k \geqslant 0} n_k 2^k \text{ et } m = \sum_{k \geqslant 0} m_k 2^k$$

où les $(n_k)$ et les $(m_k)$ sont à valeurs dans $\{0,1\}$. On définit alors une loi de composition interne sur $\mathbb{N}$, noté abusivement $\oplus$, par :

$$n \oplus m = \sum_{k \geqslant 0} (n_k \oplus m_k)\, 2^k$$

et on admet que $(\mathbb{N}, \oplus)$ est un groupe commutatif.

**Q26** Déterminer l'élément neutre de $(\mathbb{N}, \oplus)$.

**Q27** Si $x \in \mathbb{N}$, quel est l'inverse $x^{-1}$ de $x$ ?

**Q28** Soient $x_1, x_2, \ldots, x_{n-1}$ et $x_n$ des entiers naturels tels que $x_1 \oplus x_2 \oplus \ldots \oplus x_n = 0$. Soit $i_0 \in [\![1, n]\!]$ et soit $z \in \mathbb{N}$ avec $z \neq x_{i_0}$. On pose : $x'_i = x_i$ si $i \neq i_0$ et $x'_{i_0} = z$. Prouver que :

$$\bigoplus_{i=1}^{n} x'_i \neq 0$$

**Q29** Soient $x_1, x_2, \ldots, x_{n-1}$ et $x_n$ des entiers naturels tels que $x_1 \oplus x_2 \oplus \ldots \oplus x_n \neq 0$. Prouver qu'il existe $i_0 \in [\![1, n]\!]$ et $z \in \mathbb{N}$ avec $z < x_{i_0}$ de sorte que si on pose : $x'_i = x_i$ si $i \neq i_0$ et $x'_{i_0} = z$, on a :

$$\bigoplus_{i=1}^{n} x'_i = 0$$

(Si $\sum_{k \geqslant 0} a_k 2^k$ est le développement binaire de $x_1 \oplus x_2 \oplus \ldots \oplus x_n$, on pourra considérer :

$$k_0 = \max \{k \in \mathbb{N}, a_k \neq 0\}.)$$

Montrer que pour un tel $i_0$, on a nécessairement $z = (\bigoplus_{i=1}^{n} x_i) \oplus x_{i_0}$.

Une configuration du jeu est entièrement caractérisée par la donnée du nombre d'allumettes de chacun des tas. On code donc une configuration par un $k$-uplet d'entiers naturels ( $N_1, N_2, \ldots, N_k$ ). On dit d'une configuration qu'elle est favorable lorsque $N_1 \oplus N_2 \oplus \ldots \oplus N_k \neq 0$; sinon on dit qu'elle est défavorable. On note $A$ et $B$ les deux joueurs.

**Q30** Décider si la configuration $(1, 3, 5, 7)$ est favorable. Si c'est le cas, déterminer tous les coups qu'il est possible de jouer et qui placent le jeu dans une configuration défavorable.

**Q31** Même question avec $(1, 3, 4, 7)$.

**Q32** Le joueur $A$ récupère la main et le jeu est dans une configuration favorable. Comment doit jouer $A$ pour être certain de gagner la partie?

**Q33** Le joueur $A$ récupère la main et le jeu est dans une configuration défavorable. Que peut-il faire ?