Corrigé du DS2

① Arbres couvrants

Q1. $\forall s \in S_n$, $s \in C_s$ donc $C_s \neq \emptyset$

$\forall s \in S_n$, $C_s \subset S_n$ donc $\bigcup_{s \in S_n} C_s \subset S_n$.

Réciproquement $\forall s \in S_n$, $s \in C_s \subset \bigcup_{s \in S_n} C_s$ donc $S_n \subset \bigcup_{s \in S_n} C_s$.

On a bien :

$$S_n = \bigcup_{s \in S_n} C_s$$

Si $C_s \cap C_t \neq \emptyset$ alors soit $x \in C_s \cap C_t$, par définition il existe un chemin entre s et x et un chemin entre x et t, en concaténant ces chemins on obtient un chemin entre s et t. On va en déduire que $C_s = C_t$ par double inclusion.

Soit $y \in C_s$ alors en concaténant un chemin de y à s avec un chemin de s à t on obtient un chemin de y à t et donc $y \in C_t$. On a $C_s \subset C_t$.

Par symétrie des rôles, on aura aussi $C_t \subset C_s$ et donc $C_s = C_t$

Q2. Si $t \in C_s$ alors l'ensemble des chemins entre s et t est non vide et l'ensemble des longueurs des chemins entre s et t est une partie non vide de $\mathbb{N}$ qui admet donc un plus petit élément qui correspond à un plus court chemin entre s et t. Considérons c un tel plus court chemin entre s et t, si c admet deux sommets identiques, alors il admet un cycle et on pourrait en extraire un chemin c' de longueur strictement plus courte ce qui contredit la minimalité de c. Ainsi c est bien élémentaire.

Q3. a_k n'est pas une arête de G_k donc si ses extrémités sont dans une même composante connexe de G_k alors il existe dans G_k et donc dans G un chemin entre ces deux extrémités qui n'utilise pas a_k et qui ~~serait~~ donnerait lieu à un cycle dans G en ajoutant a_k - ce qui est exclu puisque G est un arbre.

On remarque que G_0 n'ayant pas d'arête possède n CC et le résultat ci-dessus garantit qu'à chaque ajout d'arête, on fusionne deux CC et donc qu'on a exactement une CC de moins.

Autrement dit, si on note n_k le nb de CC de G_k pour chaque $k \in \{0, \dots, m\}$ on remarque que $n_0 = n$
et $n_{k+1} = n_k - 1$

Ainsi $n_m = n - m$ et doit valoir 1 car G est connexe. On en déduit que

$$\underline{m = n - 1}$$

Q4. $1 \Rightarrow 2$ et $1 \Rightarrow 3$ d'après la question 3.
On remarque que compte tenue de la définition d'un arbre, il suffit de montrer que $2 \Leftrightarrow 3$.

$2 \Rightarrow 3$ Soit G connexe avec $m = n - 1$.
Par l'absurde, si G possède un cycle alors on peut lui retirer une arête tout en préservant la connexité et obtenir G' connexe avec $m' = m - 1 = n - 2$. En réitérant le

processus tant que l'on a l'existence d'un cycle on aboutirait à un graphe connexe et acyclique $\tilde{G}$ avec $\tilde{m} < n-1$ ce qui est impossible d'après la Q3.

$3 \Rightarrow 2$ Si G est acyclique avec $m = n-1$.

Soit $C_1 \dots C_t$ ses CC ayant $m_1, \dots, m_t$ arêtes chacunes et $n_1, \dots n_t$ sommets.

Chaque C_i est acyclique et donc un arbre.

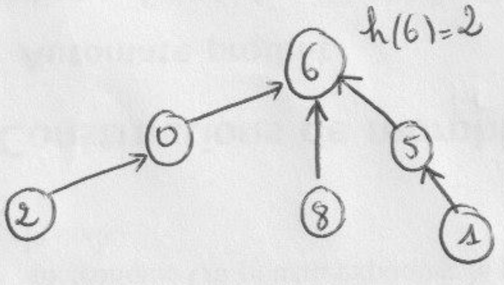
Ainsi, $\forall i \in [\![1, t]\!], \; m_i = n_i - 1$

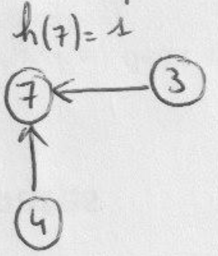
En sommant :

$$n = \sum_{i=1}^{t} n_i = \sum_{i=1}^{t} (1 + m_i) = m + t = n - 1 + t$$

On en déduit que $t = 1$ et donc que G est connexe.

Q5.





Q6.

```
int representant(int* tab, int size, int s) {
    if (tab[s] < 0) return s;
    else return (representant(tab, size, tab[s]));
}
```

~~Q7. Compression des c~~

Q7.

```c
void union(int* tab, int size, int s, int t){
    int rs = représentant (tab, size, s);
    int rt = representant (tab, size , t);
    if(rs!=rt) {
        if (tab[rs] < tab[rt]) {
            tab[rt]=rs;
        }
        else if (tab[rs]> tab[rt])
            tab[rb]=rt;
        else { tab[rs]= rt;
            tab[rt] = tab[rt] - 1;
        }
    }
}
```

Q8. On montre le résultat par récurrence sur le nombre d'appels à l'opération union.

- Si on n'a fait aucun appel alors on a $h(s)=0$ pour chaque représentants et $|X|=1$ pour chaque classe X donc c'est bien vérifié.
- Si on fait une union sur une partition qui satisfait l'hypothèse de récurrence alors le résultat reste valable pour les classes non fusionnées.

Si deux classes X et Y de représentants x et y sont fusionnées alors la classe obtenue a pour cardinal $|X|+|Y|$ et la hauteur est soit $h(x)$ soit $h(y)$ soit $h(x)+1$ (soit $h(y)+1=h(x)+1$) suivant le résultat de la comparaison de $h(x)$ et de $h(y)$.

Si $h(x) < h(y)$ alors le représentant est y et h n'est pas modifiée et on a $|X|+|Y| \geq |Y| \geq 2^{h(y)}$

Si $h(x) > h(y)$: idem - symétrique.

Si $h(x) = h(y)$ alors le nouveau rep a pour valeur de h : $h(x)+1 = h(y)+1$ et

$$|X|+|Y| \geq 2^{h(x)} + 2^{h(y)} = 2^{1+h(x)} = 2^{1+h(y)}$$

ce qui clot la récurrence.

Q9. On aura $\forall X \in \mathcal{P}$, $|X| \geq 2^{h(s)}$ et $|X| \leq n$.

Ainsi, pour tt représentant s, $h(s) \leq \log_2(n)$ et le maximum des $h(s)$ correspond à la complexité des fonctions représentant et union qui est donc en $\underline{O(\log_2(n))}$.

Q10. Compression des chemins

Q11.

```
bool est_un_arbre (graphe g) {
    int n = g.size;
    int* d = g.degrees;
    int m = 0;
    for (int i=0 ; i<n; i++) {
        m += d[i];
    }
    if (m!=(n-1)*2) return false;
    int* uf = malloc (n * sizeof(int));
    for (int i=0; i<n; i++) {
        uf[i] = -1;
    }
```

on commence par compter le nombre d'arêtes via les degrés.

on crée une structure partition

```
for (int i=0; i<n; i++) {
    for (int j=0; j<d[i]; j++) {
        union(uf, n, i, g.adj[i][j]);
    }
}
int cpt = 0;
for (int i=0; i<n; i++) {
    if (uf[i]<0) cpt++;
}
return (cpt==1);
}
```

on fusionne X_i et X_j dès que $(i,j) \in A$

on compte le nb de rep

Q12. Pour chaque sommet il n'y a qu'une seule case dans le tableau et donc un seul suivant possible, ainsi il ne peut pas y avoir deux fois un m̂ sommet x dans le chemin car on ne peut pas lui associer un suivant pour chacune de ses occurences.

Q13.

```
void greffer(arbre* a, chemin* c) {
    int sj = c->debut;
    while(sj != c->fin) {
        a->peres[sj] = c->suivant[sj];
        sj = c->suivant[sj];
    }
```

Q14.

```
chemin* chemin_aleatoire(graphe g, arbre* a, int s) {
    chemin* c = malloc(sizeof(chemin));

    c->debut = s;
    c->fin = s;
    c->size = g.size;
    c->suivant = malloc(c->size * sizeof(int));
    int dernier = s;
    while (a->peres[dernier] == -2) {
        int u_ind = rand() % (g.degrees[dernier]);
        int u = g.adj[dernier][u_ind];
        int indice = cherche(c, u);
        if (indice != -1) {
            c->fin = u; dernier = u;
        }
        else { c->suivant[dernier] = u;
            c->fin = u; dernier = u;
        }
    }
    return c;
}
```

Rmq: la variable dernier est inutile car coïncide systématiquement avec c->fin.

La fonction cherche va renvoyer l'indice de u dans c s'il est présent et -1 sinon, en fait la valeur de l'indice n'a pas d'importance et un booléen suffit.

```
int cherche (chemin* c, int u) {
    int s = c -> debut;
    int indice = 0;
    while(s != c -> fin) {
        if(s == u) return indice;
        s = c -> suivant [s];
        indice ++;
    }
    if (s == u) return indice;
    return (-1);
}
```

Q15.

```
arbre* wilson (graphe g, int r) {
    arbre* a = malloc (sizeof(a));
    a -> size = g.size;
    for (int i = 0; i < a -> size; i++) {
        a -> peres [i] = -2;
    }
    a -> peres [r] = -1;
    for (int s = 0; s < g.size; s++) {
        if (a -> peres [s] == -2) {
            chemin* c = chemin_aleatoire(g, a, s);
            greffer (a, c);
            libere (c);
        }
    }
    return a; }
```

On a besoin d'une fonction qui libère le chemin :

```
void libere (chemin* c){
        free (c->suivant);
        free (c);
}
```

Résiduels d'un langage

Q16: $x \in (uv)^{-1}L$ ssi $uvx \in L$

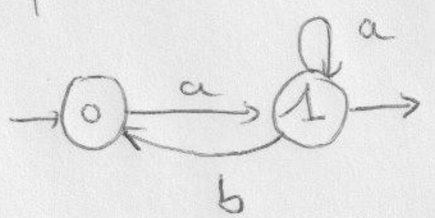
ssi $vx \in u^{-1}L$

ssi $x \in v^{-1}(u^{-1}L)$

Ainsi, $(uv)^{-1}L = v^{-1}(u^{-1}L)$

Q17

On propose



On va montrer par récurrence sur $|u|$ que $u^{-1}L$ peut être $\emptyset$, L ou $\mathcal{L}((a+ba)^*)$

Init: $\varepsilon^{-1}L = L$ $\quad a^{-1}L = \mathcal{L}((a+ba)^*)$

$b^{-1}L = \emptyset$

Hér: Soit $u \in \Sigma^{n+1}$ avec $n \in \mathbb{N}$ pour lequel l'énoncé est supposé valide.

→ si u commence par b alors $u^{-1}L = \emptyset$

→ si $u = a^{n+1}$ alors $u^{-1}L = \mathcal{L}((a+ba)^*)$

$\rightarrow$ sinon il existe $k \geq 1$ et $u' \in \Sigma^{\leq n}$ t.q

$$u = a^k b u' \text{ et } \delta^*(0, a^k b) = 0$$

donc $v \in u^{-1}L$ ssi $uv \in L$

ssi $a^k b u' v \in L$

ssi $u' v \in L$

ssi $v \in (u')^{-1}L$

Ainsi par HR, $u^{-1}L = (u')^{-1}L = \emptyset$ ou L ou $\mathcal{L}((a+ba)^*)$.

Q 18.

1. $u^{-1}L$ est un ensemble qui peut correspondre au résiduel d'un autre mot $\neq$.

Ainsi on pourrait avoir $u^{-1}L = v^{-1}L$ avec $u \neq v$ et $(u\alpha)^{-1}L \neq (v\alpha)^{-1}L$ ce qui ne définirait pas $\delta(u^{-1}L, \alpha)$ sans ambiguité.

Il est donc ici nécessaire de prouver que

$$u^{-1}L = v^{-1}L \Longrightarrow \forall \alpha \in \Sigma, (u\alpha)^{-1}L = (v\alpha)^{-1}L$$

Or on a vu à la q° 16 que $(u\alpha)^{-1}L = \alpha^{-1}(u^{-1}L)$ ce qui garantit le résultat.

On aurait pu poser alternativement:

$\forall P \in R, \forall \alpha \in \Sigma \quad \delta(P, \alpha) = \alpha^{-1}P$ sans passer par l'utilisation de u.

2. On peut montrer par rec sur $|m|$ que $\forall m \in \Sigma^*, \forall u \in \Sigma^*$

$$\delta^*(u^{-1}L, m) = (um)^{-1} L$$

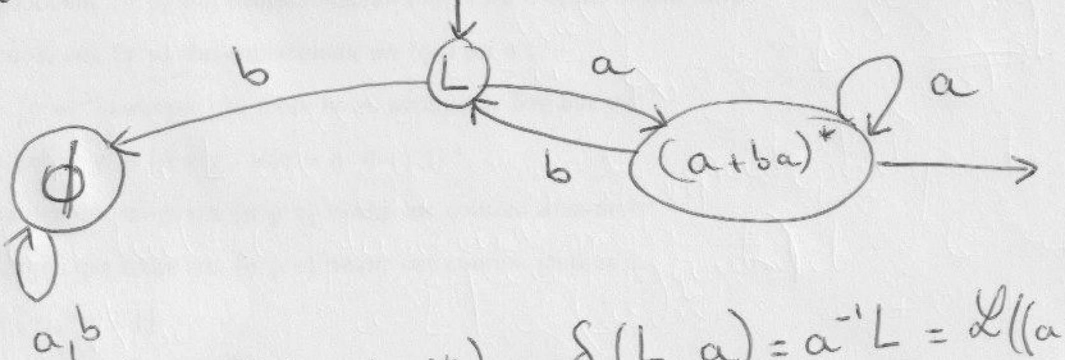
3. Ainsi $m \in L$ ssi $\varepsilon m \in L$

ssi $\varepsilon \in m^{-1}L$

ssi $\delta^*(L, m) = m^{-1}L$ contient ε

On peut donc prendre L comme état initial et $F = \{u^{-1}L \in R \text{ t.q. } \varepsilon \in u^{-1}L\}$.

4. on a 3 résiduels : $\emptyset$, L et $\mathcal{L}((a+ba)^*)$



$\delta((a+ba)^*, a) = ((a+ba)^*)$

$\delta((a+ba)^*, b) = L$

$\delta(L, a) = a^{-1}L = \mathcal{L}((a+ba)^*)$

$\delta(L, b) = b^{-1}L = \emptyset$

$\delta(\emptyset, a) = a^{-1}\emptyset = \emptyset$

$\delta(\emptyset, b) = b^{-1}\emptyset = \emptyset$

Q19.

1. $v \in u^{-1}L$ ssi $uv \in L$

ssi $\delta^*(q_0, uv) \in F$

ssi $\delta^*(\delta^*(q_0, u), v) \in F$

ssi $v \in L_{\delta^*(q_0, u)}$

2. La quest° précédente garantit que:

$$\begin{aligned} \varphi : Q &\to R \\ q &\to L_q \end{aligned} \quad \text{est surjective.}$$

En effet soit $u^{-1}L \in R$ alors $u^{-1}L = \varphi(\underbrace{\delta^*(q_0, u)}_{\text{existe car l'automate est complet.}})$

On en déduit que $|Q| \geqslant |R|$ et donc l'ensemble R est fini.

4 Le jeu de Marienbad

Q26. 0 est neutre

Q27. $x \oplus x = 0$ donc $x^{-1} = x$

Q28.

$$x_1' \oplus x_2' \oplus \cdots \oplus x_n' = x_1 \oplus \cdots \oplus x_n \underbrace{\oplus \, x_{i_0}}_{\text{permet "d'enlever" } x_{i_0} \text{ de la somme initiale.}} \oplus z$$

$$= x_{i_0} \oplus z \neq 0 \quad \text{car } x_{i_0} \neq z$$

donc ils diffèrent au moins d'un bit.

Q 29.

Soit j_0 le plus gd indice tel quee le bit d'indice j_0 de $x_1 \oplus \dots \oplus x_n$ est non nul et i_0 t.q x_{i_0} a un coefficient non nul pour le j_0^e bit.

On pose alors $z = x_1 \oplus \dots \oplus x_n \oplus x_{i_0}$

↳ Le j_0^e bit de x_{i_0} vaut 1.

↳ Le j_0^e bit de z vaut 0 car il valait 1 dans $x_1 \oplus \dots \oplus x_n$ et dans x_{i_0}.

↳ Les bits de poids $> j_0$ de x_{i_0} et z coïncident car $x_1 \oplus \dots \oplus x_n$ a une valeur nulle pour chacun d'entre eux.

<u>Ainsi, $z < x_{i_0}$</u>.

$$x_1' \oplus \dots \oplus x_n' = x_1 \oplus \dots \oplus x_n \oplus x_{i_0} \oplus z$$
$$= z \oplus z = 0$$

<u>Rmq</u>: j'ai ici appelé j_0, l'indice k_0 de l'énoncé.

Q 3~~1~~.

1 : 0 0 0 1
3 : 0 0 1 1
4 0 1 0 0
7 0 1 1 1
0 0 0 1

Configurat° favorable.
On peut par exple prendre l'allumette du 1^e paquet pour transmettre une configuration défavorable.

Q 31 . 1 : 0 0 1
3 : 0 1 1
5 : 1 0 1
7 : 1 1 1
---
0 0 0

Configurat° défavorable

Q32. Il doit identifier k_0 et un i_0 t.q le bit de N_{i_0} d'indice k_0 vaut 1. Il prend alors $x_{i_0} - (x_1 \oplus \cdots \oplus x_n \oplus x_{i_0})$ allumettes dans le tas d'indice i_0.

Q33. Il peut faire ce qu'il veut car son adversaire a de toute façon une stratégie gagnante.