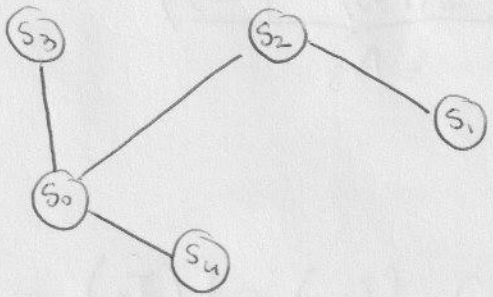
Corrigé du DS3

Q1



Q2 c'est du cours : se montre par rec sur $|S|$.

Q3 c'est du cours

Q4 Soit $G=(S,A)$ connexe

$\mathcal{P}=\{G'=(S,A') : A'\subset A \text{ et } G' \text{ est connexe}\}$

$\mathcal{P}\neq\emptyset$ car $G\in\mathcal{P}$ donc si on prend $G_0\in\mathcal{P}$ ayant un nombre minimal d'arêtes, on sait qu'il est connexe et que si on lui enlève une arête il ne le sera plus, ainsi il ne peut pas admettre de cycle car un graphe connexe avec un cycle reste connexe si on lui enlève une des arêtes de ce cycle.

G_0 est un arbre couvrant de G.

On peut ensuite en déduire l'existence d'un arbre couvrant de poids minimal car on minimise sur un ensemble fini.

Q5 (a) $T_1\neq T_2$ avec $T_1=(S,A_1)$ et $T_2=(S,A_2)$

donc $A_1\neq A_2$ avec $\#A_1=\#A_2$ donc

$A_1\not\subset A_2$ et $A_2\not\subset A_1$ càd il existe

$e\in A_2\setminus A_1$ et $e'\in A_1\setminus A_2$.

(b) Soit $H=(S,A_2\cup\{e_1\})$ H admet forcément un cycle d'après la question 2 et ce cycle contient forcément une arête $e_3\in A_3\setminus A_1$ car sinon il existerait déjà dans T_1 ce qui est impossible.

On a $f(e_1) < f(e_2) < f(e_3)$

et si on pose $T_3 = (S, \underbrace{(A_2 \setminus \{e_3\}) \cup \{e_2\}}_{= A_3})$

on a T_3 connexe

$|A_3| = |S| - 1$

et $f(T_3) = f(T_2) - f(e_3) + f(e_2) < f(T_2)$ ce qui est absurde par minimalité de T_2.

(c) On a par l'absurde, montré qu'il est impossible d'avoir $T_1 \neq T_2$ deux ACPM.

Q6. Supposons par l'absurde que $a \in B^*$ alors ~~B^*~~ $(S, B^* \setminus \{a\})$ n'est plus connexe d'après la q°2 et si on pose $a = (x, y)$, on peut considérer C_x et C_y les composantes connexes distinctes de $(S, B^* \setminus \{a\})$.

Soit $C : x = x_0 \to x_1 \to \dots \to x_n = y$ un cycle de G sur lequel a est de poids maximal.

Soit i le premier indice tel que $x_i \notin C_x$

alors $i \geq 1$ et $x_{i-1} \in C_x$

Considérons $a' = (x_{i-1}, x_i)$, on sait que $a' \notin B^*$ car sinon ses extrémités seraient dans une même composante connexe.

On pose $T^{**} = (S, \underbrace{(B^* \setminus \{a\}) \cup \{(x_{i-1}, x_i)\}}_{= B^{**}})$

on a alors $p(T^{**}) < p(T^*)$ car $p(a') < p(a)$ (maximale dans C)

$\#B^* = \#B^{**}$

et enfin, T^{**} est connexe car il admet un chemin entre x et y : $x \xrightarrow[\text{dans } B \setminus \{a\}]{} x_{i-1} \xrightarrow{a} x_i \xrightarrow[\text{dans } B^* \setminus \{a\}]{} y$ car $x_{i-1} \in C_x$ car $x_i \in C_y$

Ainsi T^{**} serait un ACPM de poids $< p(T^*)$ ce qui est exclu.

Ainsi $\underline{a \notin B^*}$

Q7
```
let rec separation l1 l2 = match l1,l2 with
  |[ ] -> [ ], [ ]
  |[a] -> [ ], [a]
  |a::b::suite -> let la, lb = separation suite in
                  a::la, b::lb
```

Q8
```
let rec fusion l1 l2 = match l1, l2 with
  |[ ], [ ] -> [ ]
  |[ ], _   -> l2
  |_, [ ]   -> l1
  |a::n1, b::n2 when a<=b -> a::(fusion n1 l2)
  |a::n1, b::n2 -> b::(fusion l1 n2)
```

Q9
```
let rec tri_fusion l = match l with
  |[ ] -> [ ]
  |[a] -> [a]
  |_ -> let l1, l2 = separation l in
        fusion l1 l2
```

Q10

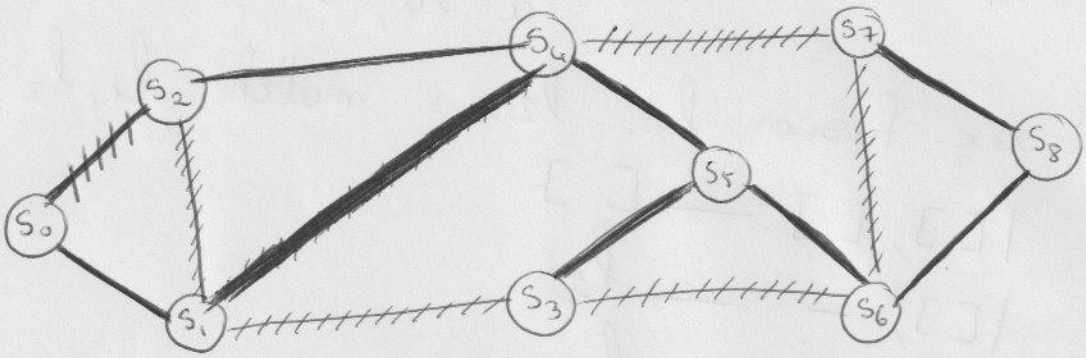
On trie les arêtes par poids croissants
On parcourt les arêtes de l'ordre obtenu.
On initialise un ensemble d'arêtes Aselec à vide.
pour chaque arête (x, y) :

si x et y sont de des CC distinctes de (S, A_{selec}) alors
$A_{selec} \leftarrow A_{selec} \cup \{(x, y)\}$

on renvoie Aselec

Complexité : $O(|A| \log |A|)$ si on implémente avec une structure union find qui gère les CC en temps quasi constant.

Q11



Q12 On va d'abord écrire une fonction qui récupère la liste des arêtes pondérées :

```
let aretes g =
    let n = Array.length g in
    let res = ref [] in
    for i = 0 to n-1 do
        List.iter (fun (p,x) -> res := (p,x,i) :: (!res)) (List.filter (fun j -> j>i) g.(i))
    done;
    !res
```

```
let aretes g = let n = Array.length g in
    let res = ref [] in
    for i = 0 to n-1 do
        List.iter (fun (j,p) -> res := (p,i,j) :: (!res)) (List.filter (fun j,p -> j>i) g.(i))
    done;
    !res
```

on ne garde que les voisins d'indices > i pour éviter les doublons.

```
let tri_aretes g =
    let l = aretes g in
    tri_fusion l
```

(comme le poids est en 1e coordonnée, la fct va bien trier en fct de la 1e coordonnée ĉ l'est rappelé par l'énoncé).

Q13 On peut faire un parcours en profondeur en ne considérant dans le voisinage que les arêtes dont le poids est strictement plus petit que poids_max.

~~let filtre x poids_max = x<<poids_max~~

```
let filtre p (x,y) -> y<p
```

```
let connectes g s t p =
    let n = Array.length g in
    let vu = Array.make n false in
    let rec visiter s =
        if (s=t) then true
        else if (vu.(s)) then false
        else
            begin
            vu.(s) <- true;
            let v = List.filter (filtre p) g.(s) in
            let lb = List.map (fun (x,y) -> visiter x) v in
            List.exists (fun x -> x) lb
            end
    in visiter s
```

Q14

```
let kruskal_inverse g =
    let copie = Array.copy g in
    let al = tri_aretes g in
    let rec aux a l = match l with
        |[] -> a
        |(p,s,t)::n when (connectes a s t p) ->
                a.(s) <- List.filter (fun (x,y) -> y<p) a.(s);
                a.(t) <- List.filter (fun (x,y) -> y<p) a.(t);
                aux a n
        |_::n -> aux a n
    in aux copie al
```

Q15 On aura pour invariant de l'algo que (S, B) est connexe. Ainsi le graphe renvoyé est connexe.
Supposons par l'absurde qu'il contient un cycle alors ce dernier a une arête de poids maximal qui aurait dû être supprimée au moment où

elle a été considérée

Ainsi le graphe renvoyé est bien connexe et acyclique, c'est un arbre.

Q16. Supposons tout d'abord f injective et posons $T^* = (S, B^*)$ l'unique arbre couvrant minimal.

Montrons que $B^* \subset B$ est un invariant de l'algo :

<u>init</u> : $B^* \subset A = B$

<u>à chaq tour</u> : si B n'est pas modifié c'est bon sinon si on décide de retirer a alors on remarque que a est dans un cycle (car $((S, B \setminus \{a\}))$ reste connexe) et est de poids maximal car sinon le cycle aurait été cassé avant. Ainsi, d'après la q°5 $a \notin B^*$ et

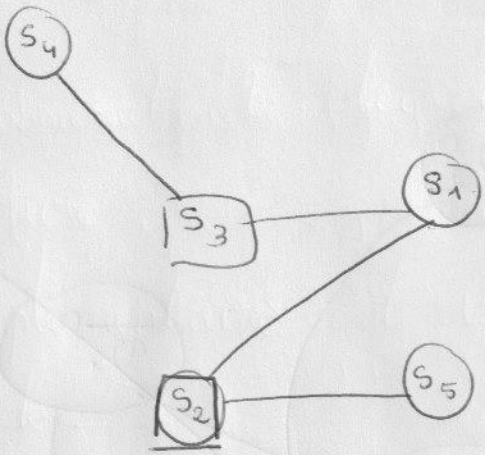
$$B^* \subset B \setminus \{a\}.$$

<u>Ccl</u> : $B^* \subset B$ avec B^* et B les arêtes d'un arbre sur les $\hat{m}$ sommets et donc de $\hat{m}$ cardinaux $\Rightarrow B^* = B$.

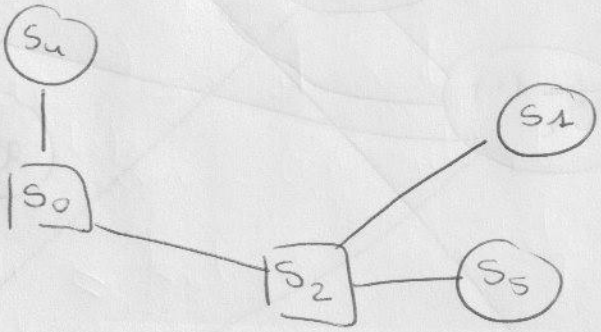
Q17 tri : $O(|A| \log |A|)$ + $|A| \times$ connecte càd $|A| \times O(|A|+|S|)$

<u>Total</u> : $O(|A|^2)$ c'est moins bien que Kruskal.

Q 18



ou



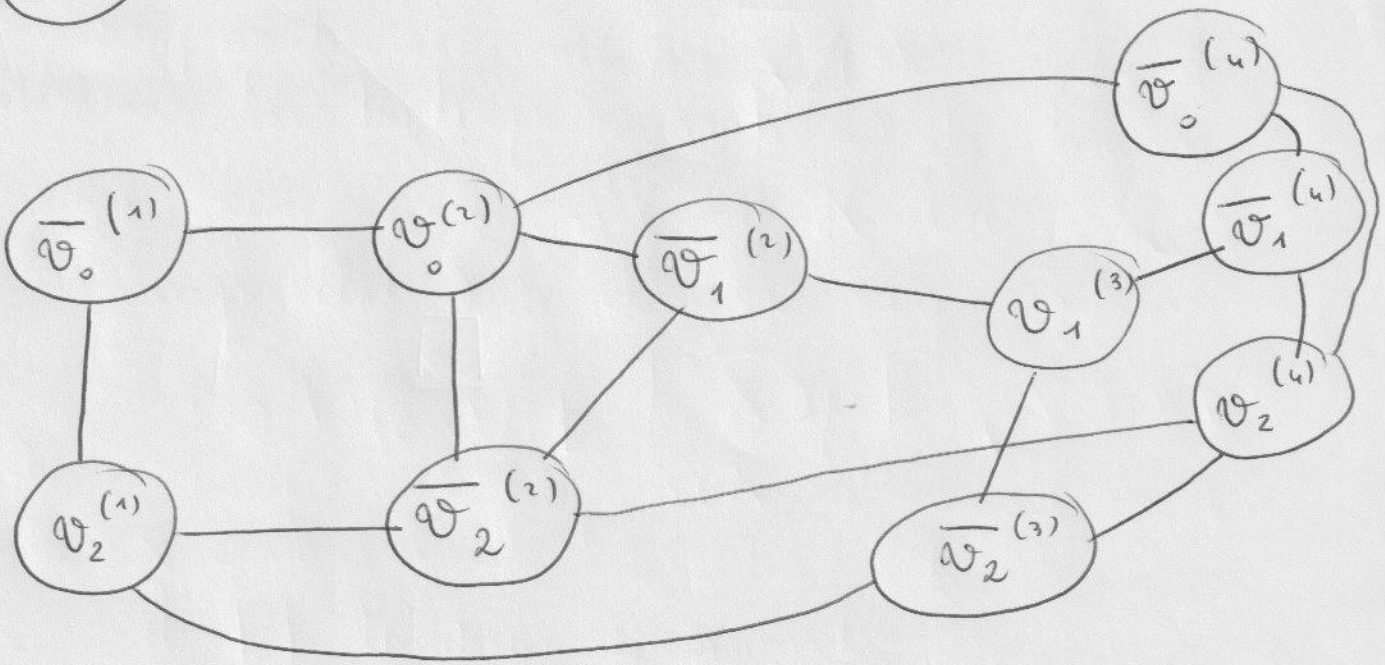
conviennent.

On remarque que tout arbre devrai contenir s_2 car c'est le seul voisin de s_5 et puisqu'il n'est pas relié à s_4, on aura besoin d'au moins un autre sommet de Steiner.

Q 19. $\{s_1, s_5, s_4\}$ est un stable de G_1. On peut représenter le graphe complémentaire pour constater qu'il ne contient pas de clique de taille 4.

Q 20 page suivante

Q20



Q21 Soit X un stable de taille m dans G_φ.

On pose v pour que chaque littéral de X soit vrai. v est bien définie car il est impossible d'avoir une variable x telle que x et $\overline{x}$ soient dans X. On pose à faux les variables qui n'apparaissent pas dans X. Ainsi, on sait que X contient au plus un sommet dans chaque triangle correspondant à une clause et puisque $|X| = m$ alors on sait qu'il en contient exactement un. Ainsi, dans chaque clause de φ, il y a bien un littéral dont le sommet est dans X et donc évalué à vrai. Ainsi $v \models \varphi$ et φ est bien satisfiable.

Q22 Soit $v \models \varphi$.

Soit $\tilde{X} = \{ v_i^{(i)} : \varphi(v_i) = \text{true} \}$.

On sait que dans chaque clause, il y a au moins un littéral à vrai et

donc dans G_4, il y a au moins un sommet de $\tilde{X}$ dans chaque triangle. X est définit en sélectionnant un elt de $\tilde{X}$ dans chaque triangle (clause) et ceci forme un stable car deux littéraux de triangles $\neq$ ayant la même valuat° ne sont pas reliés car ne peuvent pas être une variable et sa négation.

(Q23) - on peut utiliser la description d'un stable comme certificat.

- la fonct° de vérification consistera à calculer la taille de l'ensemble et à vérifier pour chaque paire d'éléments de cet ensemble qu'ils ne sont pas reliés Ceci se fait bien en tps poly en la taille de l'ensemble plus la taille du graphe.

- si un stable existe dans G alors il peut être par exemple représenté par un tableau de booléens indexé sur les sommets de G et donc en taille polynomiale en taille de G.

(Q24) - Stable $\in$ NP (q°23)

- 3SAT $\leq_p$ Stable (q°21+22)

or 3SAT est NP complet donc Stable est NP complet.

(Q25) Soit $\overline{X}$ un stable de G de cardinal k alors $S(G) \setminus \overline{X}$ est une couverture par sommets de G donc en ajoutant cet ensemble à l'ensemble X de l'énoncé on aura une couverture par sommets de G' telle que

$\forall x \in X, \exists y \in \overline{X}$ t.q $(x,y) \in A_G$.

Ainsi l'ensemble $X \cup \overline{X}$ peut être un ensemble de sommets d'un arbre de Steiner avec $\overline{X}$ comme sommets de Steiner de cardinal $|S| - k$.

Pour les arêtes de l'arbre, il suffit de prendre ~~toutes les arêtes~~ pour chaque $x \in X$, une arête (x,y) tel que y est adjacent à l'arête x puis de prendre un arbre couvrant ~~de l'arbre~~ du graphe complet comportant les sommets de G. On a clairement la connexité car chaque arête de x est reaccordée à un arbre connexe par définition et on aura exactement $\underbrace{m}_{=|A|} + \underbrace{n}_{=|S|} - 1$ arêtes avec $m+n$ sommets.

(Q26) L'arbre de Steiner comporte k sommets dans l'ensemble S.

Les sommets de X ne sont pas reliés entre eux donc chacun d'entre eux devra être relié à un

sommet de S pour garantir la connexité.
Ainsi les sommets de Steiner doivent former une couverture par sommets de taille k et leur complémentaire est alors un stable de taille $|S| - k$.

(Q27) entrées : $G = (S, A)$ et $k \in \mathbb{N}$ et $X \subset S$

sortie : true ssi il existe un arbre de Steiner avec un nb de sommets de Steiner $\leqslant k$ et de sommets terminaux X.

(Q28) - Ce pb est ds NP :

. certificat = ensemble des sommets de Steiner + ensemble d'arêtes de l'arbre qui peut être représenté avec des tableaux de booléens de tailles $|S|$ et $|A|$ qui sont polynomiales en $O(|G|)$.

. vérification : on compte le nombre de sommets de Steiner et on teste que le graphe proposé est bien un arbre par exemple en testant la connexité à l'aide d'un parcours qui est bien polynomial en la taille du graphe et en vérifiant que le nombre d'arêtes est bien un de moins que le nombre de sommets.

Q29. $s_0 - s_2 - s_1 - s_4 - s_7 - s_6 - s_8$
a une capacité de 7.

Q30 Idée T est un ACPM de (S, A, f)
ssi c'est il est de poids maximal pour $(S, A, -f)$
donc soit on applique Kruskal sur $(S, A, -f)$ soit on le modifie en parcourant les arêtes par valeurs ↘.

Q31.
Soit $\sigma = s = x_0 - x_1 - \dots - x_n = t$ un chemin de poids maximal de s à t dans T. Supposons par l'absurde qu'il existe σ^* un goulot maximal de s à t $\sigma \neq \sigma^*$

Soit a l'arête de poids minimal de σ.
Puisque T est un arbre alors $(S, B \setminus \{a\})$ n'est pas connexe. Il existe dans σ^* une arête qui relie les deux cc de $(S, B \setminus \{a\})$ notée a^*, $a^* \notin B$ et $f(a^*) \geq f(a)$ car $f(a^*) \geq c(\sigma^*) > c(\sigma) = f(a)$

Ainsi $T' = (S, (B \setminus \{a\}) \cup \{a^*\})$ est un arbre couvrant de G de poids $f(T') = f(T) + f(a^*) - f(a) > f(T)$
ce qui contredit la maximalité de T.

Précision: $(S, B \setminus \{a\})$ n'est plus connexe et notamment s et t ne seront plus de la même cc ce qui garantit que sur σ^* on peut trouver une arête reliant un sommet de la cc de s à un sommet de la cc de t.

Q32.

```
void liberer_graphe (graphe g) {
    for (int i=0; i<g.n; i++) {
        free (g.voisins[i]);
        free (g.poids[i]);
    }
    free (g.degrees);
    free (g.voisins);
    free (g.poids);
}
```

Q33 On commence par écrire un pp récursif qui prend en argument le prédecesseur.

```
void parcours (graphe T, int* pred, int s, int p) {
    if (pred[s] == -2) {
        pred[s] = p;
        for (int i=0; i<T.degres[s]; i++) {
            parcours (T, pred, T.voisins[s][i], s);
        }
    }
}
int* predecesseurs (graphe T, int s) {
    int* pred = malloc (T.n * sizeof(int));
    for (int i=0; i<T.n; i++) {
        pred[i] = -2;
    }
    parcours (T, pred, s, -1);
    return pred; }
```

Q34

```
int* goulot-max (graphe G, int s, int t, int* lc){
    graphe T = arbre-max (G);
    int *pred = predecesseurs (T, s);
    int m = 0;
    int cur = t;
    while (pred[cur] != -1) {
                    m++;
                    cur = pred[cur];

    }
    *lc = m+1;
    int* sigma = malloc ((m+1)* sizeof(int));
    cur = t;
    while (pred[cur] != -1) {
            sigma [m] = cur;
            m--;
            cur = pred[cur];
    }
    sigma [0] = s;
    free (pred);
    liberer-graphe (T);
    return (sigma);

}
```