# Composition d'informatique n°4

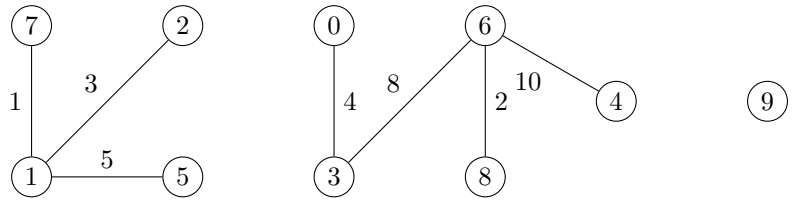
Corrigé

⋆ ⋆ ⋆

## Arbres de Steiner

## 1 Arbres et forêts

**Question 1** On obtient :




**Question 2** Le principe est le suivant :

**Entrée :** Graphe pondéré $G = (S, A, f)$ non orienté connexe
**Début algorithme**
  $B \leftarrow \varnothing$
  **Pour** $a \in A$ par ordre croissant de poids **Faire**
    **Si** $(S, B \cup \{a\})$ est sans cycle **Alors**
      $B \leftarrow B \cup \{a\}$
  **Renvoyer** $(S, B)$

En raison du tri des arêtes par poids croissant, avec une structure Union-Find suffisamment efficace ($\mathcal{O}(\log |S|$ suffit pour les différentes opérations), on obtient une complexité totale en $\mathcal{O}(|S| + |A| \log |S|)$ (le terme $|S|$ vient de la création de la structure Union-Find).

**Question 3** Dans un premier temps, comme « $(S, B)$ est sans cycle » est un invariant de boucle dans l'algorithme, ce dernier renvoie nécessairement une sous-forêt de $G$. Notons $F = (S, B)$ la forêt renvoyée et montrons qu'il s'agit d'une forêt couvrante de poids minimal. Pour ce faire, remarquons qu'il suffit de montrer que pour $X$ une composante connexe de $G$, alors $F[X]$ est un arbre couvrant de poids minimal de $G[X]$.

Par ailleurs, au cours de l'algorithme, pour $a \in A \cap \mathcal{P}_2(X)$, $(S, B \cup \{a\})$ est sans cycle si et seulement si $(X, (B \cap \mathcal{P}_2(X)) \cup \{a\})$ est sans cycle (car l'arête $a$ ne peut créer de cycle que au sein de la composante connexe $G[X]$).

Finalement, on remarque qu'en ne considérant que les arêtes de $A \cap \mathcal{P}_2(X)$ au cours de l'algorithme, elles seront considérées par ordre croissant de poids, et ajoutées à $F[X]$ que si elles ne créent pas de cycle. Cela est équivalent au fait d'appliquer l'algorithme de Kruskal à $G[X]$, ce qui permet de conclure.

**Question 4** Pour simplifier la preuve, on peut supposer que $F$ est un arbre, quitte à se restreindre à la composante connexe qui contient l'arête considérée.

Soit $a = \{s, t\}$ l'arête de poids maximal d'un cycle $C$. Supposons par l'absurde que $a \in B^*$. Alors $(S, B^* \backslash \{a\})$ contient deux composantes connexes $X_1$ et $X_2$.

De plus, il existe une arête $a'$ du cycle $C$ qui relie $X_1$ et $X_2$. En effet, si le cycle s'écrit $(s = s_1, s_2, s_2, \ldots, s_k = t)$, alors il existe un indice $i \in [\![1, k-1]\!]$ minimal tel que $s_i \in X_1$ et $s_{i+1} \in X_2$ (car $s_1 \in X_1$ et $s_k \in X_2$ par hypothèse).

On pose alors $a' = \{s_i, s_{i+1}\}$ et $F' = (S, B^* \cup \{a'\} \setminus \{a\})$. $F'$ est sans cycle (on a rajouté une arête entre deux composantes connexes) et possède $|S| - 1$ arêtes (car $F$ était un arbre). C'est donc un arbre. Pourtant, $f(F') = f(F) + f(a') - f(a) < f(F)$, ce qui est absurde par minimalité de $F$. On conclut que $a \notin B^*$.

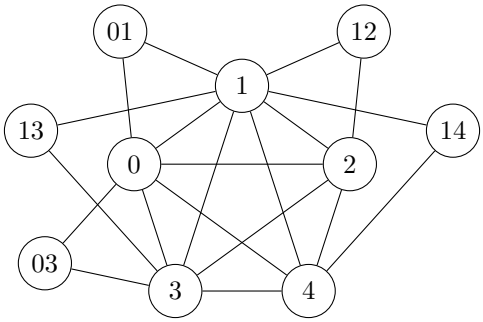
**Question 5** Comme précédemment, on peut se restreindre au cas où $F$ est un arbre.

Soit $X \subsetneq S$ et $a = \{s, t\}$, $s \in X$, l'arête de poids minimal parmi celles ayant exactement une extrémité dans $X$. Supposons $a \notin B^*$ et soit $(s = s_1, s_2, \ldots, s_k = t)$ un chemin de $s$ à $t$ dans $F$. Comme précédemment, il existe une arête $a' = \{s_i, s_{i+1}\}$ telle que $s_i \in X$ et $s_{i+1} \notin X$. Alors $F' = (S, B^* \cup \{a\} \setminus \{a'\})$ est un arbre (il est connexe car on a supprimé une arête d'un cycle et possède $|S| - 1$ arêtes) de poids $f(F') = f(F) + f(a) - f(a') < f(F)$, ce qui est absurde.

# 2 Problème de l'arbre de Steiner

**Question 6** Pour une instance positive $(G = (S, A), X, k)$ de ADS, un certificat est un ensemble $Y \subseteq S$ (donc de taille polynomiale). On peut vérifier en temps polynomial que $X \subseteq Y$, que $G[Y]$ est connexe (donc admet un arbre couvrant) et que $|Y| \leqslant k + 1$ (donc un arbre couvrant de $G[Y]$ possède au plus $k$ arêtes).

**Question 7** On obtient le graphe :




**Question 8** On procède par double implication.

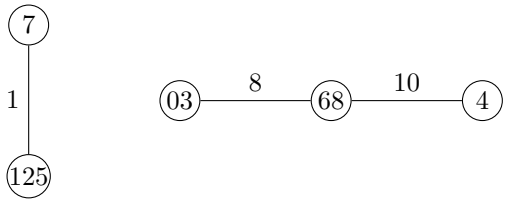
- Soit $C$ une couverture par sommets de $G$ de taille $k$. Alors pour tout $a = \{s, t\} \in A$, $s \in C$ ou $t \in C$. On en déduit que dans $G_I$, chaque élément de $A$ est adjacent à un élément de $C$. Sachant que tous les éléments de $S$ sont adjacents dans $G_I$, le sous-graphe induit par $C \cup A$ est connexe dans $G_I$ et d'ordre $|A| + k$. Il existe donc un arbre couvrant de ce sous-graphe. Cet arbre est bien un arbre de Steiner couvrant $A$ et avec $|A| + k - 1$ arêtes.
- Soit $T = (Y, B)$ un arbre de Steiner couvrant $A$ dans $G_I$, avec $|B| = |A| + k - 1$. Comme c'est un arbre, il possède $|A| + k$ sommets, soit $k$ sommets appartenant à $S$ en particulier. Posons $C = Y \setminus A$. Alors pour $a \in A$, $a$ possède un voisin $x$ dans $T$. Comme les éléments de $A$ ne sont pas adjacents dans $G_I$, on en déduit que $x \in C$. Par définition du graphe d'incidence, cela signifie que $x \in a$. Ainsi, $C$ est bien une couverture par les sommets de $G$ de cardinal $k$.

**Question 9** La construction de $G_I$ peut bien se faire en temps polynomial en la taille de $G$. On en déduit que CPS $\leqslant_{\mathsf{m}}^{\mathsf{p}}$ ADS. Comme CPS est NP-complet, on en déduit que ADS est NP-difficile. Par ailleurs, ADS $\in$ NP. On en déduit que ADS est NP-complet.

# 3 Calcul linéaire d'une forêt couvrante minimale

## 3.1 Phase de Borůvka

**Question 10** On obtient :




**Question 11** On se contente de parcourir les arêtes pour créer les mêmes dans le graphe avec mémoire. La fonction `traiter` ajoute une arête au nouveau graphe. On l'applique à la liste d'adjacence de $s$.

```
let convertir g =
    let n = Array.length g in
    let g_mem = Array.make n [] in
    for s = 0 to n - 1 do
        let traiter (t, p) =
            let v = {nom = t; poids = p; origine = (s, t)} in
            g_mem.(s) <- v :: g_mem.(s) in
        List.iter traiter g.(s)
    done;
    g_mem;;
```

**Question 12** En appliquant le résultat de la question 5 (propriété de coupe) au singleton $\{s\}$ pour chaque $s \in S$, on en déduit que pour tout $s \in S$, $\{s, v_{\min}(s)\} \in B^*$, ce qui conclut.

**Question 13** Soient $C_0, \dots C_{m-1}$ les composantes connexes de $G$. Alors $\sum_{i=0}^{m-1} |C_i| = |S|$. On en déduit que :

$$2|S'| = 2 \sum_{\substack{i=0 \\ |C_i| \geqslant 2}}^{m-1} 1 \leqslant \sum_{\substack{i=0 \\ |C_i| \geqslant 2}}^{m-1} |C_i| \leqslant \sum_{i=0}^{m-1} |C_i| = |S|$$

**Question 14** On a ici deux cas de base, selon que la liste soit vide ou non.

```
let rec voisin_min = function
    | []  -> {nom = -1; poids = -1; origine = (-1, -1)}
    | [v] -> v
    | v :: q -> let w = voisin_min q in
                if v.poids < w.poids then v else w;;
```

**Question 15** On commence par calculer un tableau des arêtes minimales avec la fonction précédente. Ensuite, pour chaque sommet, s'il a une arête minimale incidente qui n'a pas déjà été ajoutée (c'est ce que vérifie le test `if` dans la boucle), alors on l'ajoute à l'arbre `f`, ainsi que l'arête d'origine à l'arbre `fcm` (c'est ce que fait la fonction `ajout_arete`).

```
let foret_boruvka g fcm =
    let n = Array.length g in
    let f = Array.make n [] in
    let aretes = Array.map voisin_min g in
    let ajout_arete s v =
        let sori, tori = v.origine in
        fcm.(sori) <- (tori, v.poids) :: fcm.(sori);
        fcm.(tori) <- (sori, v.poids) :: fcm.(tori) ;
        f.(s) <- (v.nom, v.poids) :: f.(s);
        f.(v.nom) <- (s, v.poids) :: f.(v.nom) in
    for s = 0 to n - 1 do
        let v = aretes.(s) in
        if v.nom <> -1 && (aretes.(v.nom).nom <> s || s < v.nom) then
            ajout_arete s v
    done;
    f;;
```

**Question 16** On commence par créer la table de hachage et à ajouter chaque voisin, en utilisant le nom comme clé et le voisin comme valeur, en ne conservant que l'association de poids minimal. Ensuite, on itère sur toutes les associations pour créer une liste des voisins conservés.

```
let elim_doublons lst =
    let ht = Hashtbl.create 1 in
    let traiter v =
        if not (Hashtbl.mem ht v.nom) ||
           (Hashtbl.find ht v.nom).poids > v.poids then
            Hashtbl.replace ht v.nom v in
    List.iter traiter lst;
    let lst_elim = ref [] in
    Hashtbl.iter (fun _ v -> lst_elim := v :: !lst_elim) ht;
    !lst_elim;;
```

**Question 17** Une fonction classique de parcours de graphe. On pense à itérer sur tous les sommets et à marquer les sommets isolés.

```
let composantes g f =
    let n = Array.length g in
    let comp = Array.make n (-1) and m = ref 0 in
    let rec dfs s =
        if comp.(s) = -1 then begin
            comp.(s) <- !m;
            List.iter (fun (t, p) -> dfs t) f.(s)
        end in
    for s = 0 to n - 1 do
        if comp.(s) = -1 then
            if g.(s) = [] then comp.(s) <- -2
            else (dfs s; incr m)
    done;
    comp, !m;;
```

**Question 18** On commence par calculer la forêt, en mettant à jour `fcm` grâce à la fonction `forêt_boruvka`. Ensuite, pour chaque arête de $G$, si elle est entre deux composantes connexes, on l'ajoute au nouveau graphe avec mémoire, dont les sommets sont les composantes. Enfin, on élimine les arêtes en doublons. Comme les poids sont tous distincts, on est sûr de garder la même arête entre deux composantes lorsqu'on fera un appel

à `elim_doublons` sur les deux listes d'adjacence.

```
let phase_boruvka g fcm =
    let n = Array.length g in
    let f = foret_boruvka g fcm in
    let comp, m = composantes g f in
    let cfg = Array.make m [] in
    for s = 0 to n - 1 do
        let ajout_arete v =
            if comp.(s) <> comp.(v.nom) then
                let v' = {nom = comp.(v.nom); poids = v.poids; origine = v.origine} in
                cfg.(comp.(s)) <- v' :: cfg.(comp.(s)) in
        List.iter ajout_arete g.(s)
    done;
    for c = 0 to m - 1 do
        cfg.(c) <- elim_doublons cfg.(c)
    done;
    cfg;;
```

**Question 19** On détaille :

- `foret_boruvka` : la fonction `voisin_min` a une complexité linéaire en la taille de la liste. Ainsi, dans `foret_boruvka`, le calcul des arêtes minimales est en $\mathcal{O}(|S| + |A|)$ (on crée un tableau de taille $|S|$, en appliquant `voisin_min` à chaque liste d'adjacence d'un sommet $s$, de taille $\deg(s)$). La boucle `for` qui suit a une complexité en $\mathcal{O}(|S|)$ ;
- `composantes` est un parcours de graphe en $\mathcal{O}(|S| + |A|)$ ;
- `elim_doublons` a une complexité linéaire en la taille de la liste en moyenne (car les opérations sur les tables de hachages sont constantes en moyenne).

Finalement, `phase_boruvka` fait un appel à `foret_boruvka` puis à `composantes` (qui est en $\mathcal{O}(|S|)$ car le graphe auquel on l'applique est une forêt), puis parcourt chaque liste d'adjacence de $G$ pour créer les arêtes entre les composantes ($\mathcal{O}(|S| + |A|)$), avant d'éliminer les doublons (à nouveau $\mathcal{O}(|S| + |A|)$).

La complexité totale est en $\mathcal{O}(|S| + |A|)$.

## 3.2 Sélection aléatoire d'arêtes

**Question 20** On se contente de parcourir les arêtes et de les garder avec une probabilité $\frac{1}{2}$. Attention, pour être sûr qu'une arête apparaisse bien dans les deux listes d'adjacence concernée, on ne traite que les cas où un sommet est plus petit que son voisin.

```
let demi_graphe g =
    let n = Array.length g in
    let h = Array.make n [] in
    for s = 0 to n - 1 do
        let traiter v =
            if s < v.nom && Random.int 2 = 0 then begin
                h.(s) <- (v.nom, v.poids) :: h.(s);
                h.(v.nom) <- (s, v.poids) :: h.(v.nom);
            end in
        List.iter traiter g.(s)
    done;
    h;;
```

**Question 21** La propriété de cycle (question 4) donne directement le résultat.

**Question 22** Pour montrer ce résultat, imaginons la construction de $H$ au cours de l'algorithme version 2, pour se convaincre que la forêt $F$ renvoyée aura bien la même probabilité d'être obtenue que dans la version 1 : si on remplace les lignes 5 à 8 par :

**Algorithme :** version 2
**Entrée :** Graphe $G = (S, A, f)$ pondéré non orienté
1 **Début algorithme**
2  Poser $B = \emptyset$ et $B' = \emptyset$.
3  **Pour** chaque arête $a \in A$ par poids croissant **Faire**
4  $x \leftarrow$Tirer à pile ou face.
5  **Si** $(S, B \cup \{a\})$ ne contient pas de cycle **Alors**
6  **Si** $x$ vaut pile **Alors**
7  $B \leftarrow B \cup \{a\}$
8  **Si** $x$ vaut pile **Alors**
9  $B' \leftarrow B' \cup \{a\}$
10  **Renvoyer** $F = (S, B)$.

Alors la forêt $F$ obtenue n'a pas changé. De plus, par principe de l'algorithme de Kruskal, $F$ est une forêt couvrante minimale de $H = (S, B')$. Enfin, le graphe $H$ est bien un graphe $\frac{1}{2}G$ par construction, ce qui permet de conclure.

**Question 23** Les arêtes étant considérées par poids croissant, si $(S, B \cup \{a\})$ contient un cycle, sachant que $(S, B)$ n'en contient pas, cela signifie bien que $a$ est une arête de poids maximal dans un cycle créé par son ajout à $F$. Réciproquement, si $(S, B \cup \{a\})$ ne contient pas de cycle, alors on distingue : soit on a tiré pile, auquel cas $a$ apparaît dans $F$, donc est $F$-légère, soit on a tiré face, auquel cas $a$ n'apparaît pas dans $F$, mais si elle crée un cycle lorsqu'on l'ajoute à $F$, alors ce cycle contient une arête plus lourde que $a$ (sinon elle aurait déjà été présente dans l'arbre au moment du test).

**Question 24** On remarque que dans la version 2, une arête n'est ajoutée à $B$ que si on a obtenu un pile. Comme une forêt à $n$ sommets ne peut contenir qu'au plus $n-1$ arêtes (sinon le graphe contient un cycle), cela signifie qu'on a obtenu strictement moins de $n$ piles. Enfin, puisqu'on fait un tir de pile ou face pour chaque arête $F$-légère (d'après la question précédente), le nombre $Y$ de tirs à pile ou face effectués (qui correspond au nombre d'arêtes $F$-légères) vérifie $\mathbb{E}(Y) < 2n$, ce qui conclut.

## 3.3 Calcul de forêt couvrante minimale

**Question 25** On suit la description de l'algorithme. On fait appel à l'une ou à l'autre des deux fonctions selon qu'on souhaite une mémoire ou non.

```
let rec calcul_fcm g =
    let n = Array.length g in
    if n = 1 then g
    else begin
        let fcm = Array.make n [] in
        maj_fcm (convertir g) fcm;
        fcm
    end
and maj_fcm g fcm =
    if Array.length g > 1 then
        let g' = phase_boruvka (phase_boruvka g fcm) fcm in
        let h = demi_graphe g' in
        let f' = calcul_fcm h in
        maj_fcm (alleger g' f') fcm
```

**Question 26** La question 12 garantit que les arêtes ajoutées à `fcm` doivent bien être présentes dans la forêt couvrante minimale. La question 21 garantit que les arêtes supprimées du graphe ne doivent pas apparaître dans la forêt couvrante minimale. Ainsi, au cours de l'appel `maj_fcm g fcm`, la propriété « les arêtes de $F^*$ sont contenues dans l'union des arêtes de `g` et celles de `fcm` » est vraie à tous les appels récursifs. Comme la fonction termine lorsque $g$ ne contient plus d'arêtes, cela permet bien de conclure (car `fcm` est toujours une forêt).

**Question 27** La complexité totale est la somme des complexités de tous les appels récursifs. Or, l'ensemble des fonctions autres que ces deux ont une complexité linéaire en la somme du nombre de sommets et d'arêtes, ce qui permet de conclure.

**Question 28** On commence par remarquer que dans un appel à l'algorithme, le graphe $G'$ a un nombre de sommets qui est divisé par 4. Ainsi, on peut montrer par récurrence qu'un nœud à profondeur $k$ correspondra à un graphe avec au plus $\frac{|S|}{4^k}$ sommets. Ainsi, comme le cas de base est le cas $|S| = 1$, l'arbre sera de hauteur au plus $\log_4 |S|$.

Par ailleurs, si un nœud interne est un graphe $G_k = (S_k, A_k)$, alors dans le pire cas (cas où on ne supprime aucune arête $F'$-lourde), chaque arête de $A_k$ peut se retrouver :

- dans l'enfant gauche si l'arête a été choisie dans la construction de $H$ et n'est pas dans $F'$ ;
- dans l'enfant droit si l'arête n'a pas été choisie dans la construction de $H$ ;
- dans les deux enfants si l'arête fait partie de la forêt couvrante minimale $F'$ de $H$ ;
- dans aucun enfant si l'arête a été supprimée lors des phases de Borůvka.

Or, le nombre d'arêtes de $F'$ est au plus $|S'| - 1 \leqslant \frac{|S|}{4}$, et le nombre d'arêtes supprimées lors des phases de Borůvka est au moins $\frac{|S|}{2}$. Ainsi, on peut montrer par induction que le nombre total d'arêtes à une profondeur $k$ fixée est au plus $|A|$. De même, par ce qui a été dit plus haut, le nombre total de sommets à une profondeur $\pi$ fixée est au plus $\frac{|S|}{2^\pi}$.

La complexité totale est donc en $\mathcal{O}((|S| + |A|) \log |S|)$.

**Question 29** Par construction de $H$, un enfant gauche d'un graphe $G_i = (S_i, A_i)$ possède en moyenne la moitié des arêtes de $G'_i$, c'est-à-dire moins de $\frac{|A_i|}{2}$. Ainsi, le nombre d'arêtes total de la branche gauche est inférieur à $|A_1| \sum_{i=0}^{+\infty} \frac{1}{2^i} = 2|A_1|$.

**Question 30** On remarque que l'ensemble des branches gauches forme une partition des nœuds de l'arbre des appels. Il suffit donc de calculer le nombre d'arêtes total des enfants droits et de la racine.

Or, si $G_1 = (S_1, A_1)$ est l'enfant droit d'un graphe $G_0 = (S_0, A_0)$, alors par la question 24, on a $\mathbb{E}(|A_1|) \leqslant 2\frac{\mathbb{E}(|S_0|)}{4} = \frac{\mathbb{E}(|\mathbb{S}_{\not\sim}|}{2}$. Or, si $G_0$ est un nœud de profondeur $\pi$, alors $|S_0| \leqslant \frac{|S|}{4^\pi}$.

Comme il y a $2^{\pi-1}$ enfants droits de profondeur $\pi$, on en déduit que le nombre total d'arêtes de tous les graphes est en moyenne inférieur à :

$$2|A| + 2 \sum_{\pi=1}^{\log_4 |S|} 2^{\pi-1} \frac{1}{2} \frac{|S|}{4^\pi} \leqslant 2|A| + |S|$$

Comme le nombre total de sommets est borné par $2|S|$, on en déduit la complexité voulue.

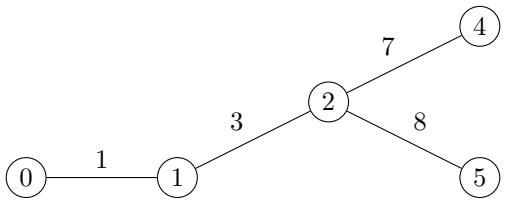
Il s'agit d'un algorithme Las Vegas (résultat toujours correct, mais temps de calcul variable).

# 4 Approximation

**Question 31** Soit $(G = (S, A), X, k)$ une instance de `ADS`. On pose $f$ une fonction de pondération de $G$ constante égale à 1. Alors $(G, X, k)$ est une instance positive de `ADS` si et seulement si la solution de `ADSP` pour

$((S, A, f), X)$ est de poids inférieur ou égal à $k$. Ainsi, si `ADSP` peut être résolu en temps polynomial, alors `ADS` aussi (car la construction de la fonction de pondération se fait en temps polynomial).

**Question 32** On obtient l'arbre :




**Question 33** Par construction, $X \subseteq Y$, et $G[Y]$ est connexe (car les sommets rajoutés forment bien des chemins entre les différents sommets de $X$). On en déduit que $T$ est un arbre qui couvre $Y$, donc en particulier couvre $X$.

**Question 34** On peut calculer les distances (et les plus courts chemins) entre les sommets de $X$ dans $G$ en temps $\mathcal{O}(\min(|S|^3, |X||A|\log|S|))$ : le premier terme correspond à l'application de l'algorithme de Floyd-Warshall au graphe $G$, et le deuxième terme à l'application de l'algorithme de Dijkstra à chaque sommet de $X$. Selon le cardinal de $X$, $S$ et $A$, l'une ou l'autre des deux méthodes peut être plus efficace.

La construction de $T_1$ peut se faire en temps $\mathcal{O}(|X|^2)$ en moyenne et $\mathcal{O}(|X|^2 \log|X|)$ dans le pire cas d'après la partie précédente.

La construction de $Y$ se fait en temps $\mathcal{O}(|X|^2|S|)$ (on considère chaque paire de sommets de $X$ et on reconstruit le chemin, de taille au plus $|S|$).

Le calcul de $T$ se fait en temps $\mathcal{O}(|S| + |A|)$ en moyenne et $\mathcal{O}(|A|\log|S|)$ dans le pire cas.

Au total, en majorant $|X|$ par $|S|$, on a une complexité en $\mathcal{O}(\min(|S|^3, |S||A|\log|S|))$ (qui est le terme dominant).

**Question 35** On commence par remarquer que $f(T) \leqslant d(B)$, car on a remplacé les arêtes de $B$ par des chemins de même poids, puis supprimé des arêtes. Dès lors, soit $T^*$ un arbre de Steiner couvrant $X$ de poids minimal. Un parcours en profondeur cyclique de $T^*$ passant deux fois par chaque arête a un poids égal à $2f(T^*)$. Par ailleurs, la somme des poids des arêtes reliant deux sommets terminaux consécutifs dans ce parcours est supérieure ou égale à la distance entre ces sommets. On en déduit que $2f(T^*)$ est supérieur ou égal au poids d'un cycle hamiltonien dans $H$ parcourant les sommets terminaux dans le même ordre. De plus, en supprimant une arête de ce cycle hamiltonien, on obtient un arbre couvrant de $H$. On en déduit $2f(T^*) \geqslant d(B) \geqslant f(T)$. L'algorithme est bien une 2-approximation.

⋆ ⋆ ⋆