# IFT800 - Série d'exercices #5

Manuel Lafond

## Décomposition en arbre

*Exercice* 1. Considérez les variantes des graphes simples avec une seule arête modifiée, et calculez la treewidth.

1. Soit $G$ un cycle avec une arête supplémentaire (i.e. $G$ est le cycle $(v_1, v_2, \ldots, v_n, v_1)$, plus une arête $v_iv_j$ quelconque). Est-ce que $tw(G) = 2$?

2. Soit $G$ un arbre avec une arête supplémentaire ajoutée. Est-ce que $tw(G) = 1$?

3. Soit $G$ une clique avec une arête supprimée. Est-ce que $tw(G) = n - 1$?

---

**Solution.**

a) Soit $G$ un cycle avec une arête supplémentaire $v_iv_j$. Considérez $G - v_i$. Ceci est un arbre et on peut obtenir une décomposition en arbre $T = (V, E)$ de $G - v_i$ de largeur 1. Si on ajoute $v_i$ à tous les sacs de $T$, la condition 2 des décopmpositions est satisfaite, car toutes les arêtes incidentes à $v_i$ sont dans un sac. En fait, $v_i$ pourrait même être voisin de tout le monde. La condition 3 des décompositions est respectée (car les sacs contenant $v_i$ forment un sous-graphe connexe). Donc $tw(G) = 2$.

b) La réponse est non. Si on ajoute une arête à un arbre, le graphe résultant a un cycle. On sait qu'un cycle a une treewidth de 2, et donc le graphe en entier doit avoir une treewidth d'au moins 2 aussi.

c) Soit $G = (V, E)$ une clique mais avec une arête $uv \notin E$. On peut obtenir une décomposition avec $n - 1$ sommets par sac, donc de taille $n - 2$. Soit $T = (V_T, E_T)$ où $V$ contient seulement deux sacs $B_u$ et $B_v$. On met $B_u =$

$V \setminus \{v\}$ et $B_v = V \setminus \{u\}$. À vous de vous convaincre que c'est bel et bien une décomposition.

---

*Exercice* 2. Soit $G$ une grille $n \times n$ (i.e. les sommets sont $v_{i,j}$ pour $i, j \in \{1, \ldots, n\}$ et il y a une arête entre $v_{i,j}$ et $v_{i+1,j}$ et entre $v_{i,j}$ et $v_{i,j+1}$ lorsque $i + 1 \leq n$ et $j + 1 \leq n$, respectivement).

Montrez que $tw(G) \leq n$. Il est recommandé de décrire une décomposition.

Si vous voulez un défi, montrez que $tw(G) = n$. Par contre, je ne donnerai pas la solution.

---

**Solution.**

On peut concevoir une dépomposition où on a le premier sac $B = \{v_{1,1}, \ldots, v_{1,n}\}$ (toute la première rangée). On le met voisin d'un deuxième sac $B' = B \cup \{v_{2,1}\}$. Ensuite, on parcourt les sommets de $G$ dans l'ordre (de gauche à droite, en descendant d'une rangée quand on est au bout). À chaque nouveau sommet $v_{i,j}$, on crée un nouveau sac avec $v_{i,j}$ et on enlève le sommet le plus ancien du sac précédent.

Plus formellement, pour chaque $i, j$ entre 1 et $n$, soit $B_{i,j}$ le sac qui contient

$$v_{i-1,j}, v_{i-1,j+1}, v_{i-1,j+2}, \ldots, v_{i-1,n}$$

ainsi que les sommets

$$v_{i,1}, v_{i,2}, \ldots, v_{i,j}$$

(si un de ces sommets n'existe pas, par exemple si $i = 1$, on l'ignore). On voit que chaque sac $B_{i,j}$ a au plus $n+1$ éléments. Donc la taille d'une décomposition avec les $B_{i,j}$ sera de $n$.

Dans $T$, on connecte $B_{i,j}$ avec $B_{i,j+1}$, ou si $j = n$, on connecte $B_{i,n}$ avec $B_{i+1,1}$. La décomposition forme donc un chemin $(B_{1,1}, B_{1,2}, \ldots, B_{n,n})$.

On peut voir que la condition 2 sera satisfaite : une arête $v_{i,j-1}v_{i,j}$ sera dans le sac $B_{i,j}$ et une arête $v_{i-1,j}v_{i,j}$ sera également dans le sac $B_{i,j}$. Pour la condition 3, on note qu'un sommet $v_{i,j}$ apparaît pour la première fois dans $B_{i,j}$, et n'apparaît que dans les $n$ sacs suivant sur le chemin $T$. Donc tous les sacs contenant $v_{i,j}$ forment un sous-graphe connexe.

---

*Exercice* 3. Soit $G = (V, E)$ un graphe. La contraction d'une arête $ab \in E$ consiste à 1) ajouter un nouveau sommet $z$; 2) mettre $N(z) = N(a) \cup N(b)$; 3) enlever $a$ et $b$ de $G$.

Soit $G'$ le graphe obtenu par la contraction d'une arête $ab$ de $G$. Montrez que $tw(G') \leq tw(G)$.

---

**Solution.**

Soit $G'$ obtenu de $G$ en contractant $ab$, résultant en un nouveau sommet $z$. Soit $T = (V_T, E_T)$ une démposition de $G$. Soit $T' = (V'_T, E'_T)$ obtenu de $G$ en remplaçant chaque occurrence de $a$ par $z$, et chaque occurrence de $b$ par $z$. On prétend que $T'$ est une décomposition de $G'$.

Pour la condition 2, on note qu'une arête $uv \in E(G')$, où $u$ et $v$ sont distincts de $z$, est toujours dans un sac de $T'$ (car $uv \in E(G)$ et $uv$ était dans un sac de $T$). Maintenant, soit une arête $zv$ de $G'$ contenant $z$. Alors dans $G$, on avait soit $av \in E(G)$ ou $bv \in E(G)$ (ou les deux). Dans les deux cas, un sac de $T$ contenait l'arête et donc un sac de $G'$ contient $zv$ car on a remplacé tous les $a$ et tous les $b$ par $z$.

Pour la condition 3, on note que dans $G$, il y a un sac $B_i$ qui contient $a$ et $b$ car $ab \in E(G)$. De plus, les sacs contenant $a$ sont connexes, et les sacs contenant $b$ sont connexes. Ces sacs se rencontrent en $B_i$, et donc les sacs contenant *$a$ ou $b$* forment un sous-graphe connexe. Donc dans $T'$, les sacs contenant $z$ forment un sous-graphe connexe.

On a donc obtenu une décomposition de $G'$ de la même taille que $T$.

---

*Exercice* 4. Soit $G = (V, E)$ un graphe. Un sous-ensemble de sommets $X \subseteq V$ est appelé un *séparateur* si $G - X$ n'est pas connexe. Soient $G_1, \ldots, G_k$ les graphes formant les composantes connexes de $G - X$. Montrez que $tw(G) \leq |X| + \max_i(tw(G_i))$.

---

**Solution.**

Soient $T_1, \ldots, T_k$ des décompositions de taille minimum des graphes $G_1 - X, \ldots, G_k - X$. Par définition, les $G_i - X$ n'ont aucun sommet en commun. Pour obtenir une décomposition pour $G$ en entier, on ajoute un sac $B_X = X$, et on ajoute $X$ à tous les sacs de tous les $T_1, \ldots, T_k$. On obtient $T$ en connectant

$B_X$ à la racine de chaque $T_i$ résultant. La taille de cette décomposition est en fait $|X| + \max_i(tw(G_i - X)) \leq |X| + \max_i(tw(G_i))$.

On va argumenter que c'est bien une décomposition. Les conditions 2 et 3 sont bien satisfaites pour toutes les arêtes $uv$ et tous les sommets $u$ qui n'impliquent pas un sommet de $X$. La condition 2 est satisfaite pour $X$ car toutes les arêtes impliquant un sommet de $X$ sont dans un sac (vu que $X$ est brutalement ajouté partout). La condition 3 est satisfaite pour tout sommet de $X$ puisque $X$ est partout.

---

*Exercice* 5. Soit $G = (V, E)$ un graphe. Soit le problème suivant.

VERTEX-COVER
**Entrée :** Graphe $G = (V, E)$
**Paramètre :** $tw(G)$
**Sortie :** un ensemble couvrant les arêtes $X \subseteq V$ de taille minimum

Supposez qu'on vous donne un décomposition en arbre $T = (V_T, E_T)$ de $G$. Donnez un algorithme de programmation dynamique en temps $O(f(tw(G))n^c)$ sur $T$ pour résoudre le problème VERTEX-COVER. Il est suffisant de décrire vos récurrences.

---

**Solution.**

On suppose qu'on nous donne une jolie décomposition $T = (V_T, E_T)$. Un coloriage d'un sac $B_i$ est une fonction $c_i : B_i \to \{vert, rouge\}$. On interprète $vert$ comme "est dans le vertex-cover", et $c_i^{vert}$ est l'ensemble des sommets verts. La notion de compatibilité est telle que définie en classe.

J'utilise la notation $V_i$ pour dénoter l'ensemble des sommets contenu dans $B_i$ ou dans un descendant de $B_i$ (ce que j'appelais $G(B_i)$ en classe). Donc, $G[V_i]$ dénote le sous-graphe de $G$ induit par $V_i$. On définit $M[c_i, B_i]$ comme le nombre minimum de sommets dans un vertex-cover $X$ de $G[V_i]$, avec la restriction que $X \cap B_i = c_i^{vert}$ (ou $\infty$ si ceci est impossible).

Soit $B_i$ une feuille. Notons qu'on doit vérifier que chaque arête présente dans le sac $B_i$ doit être couverte par au moins un sommet vert, sinon on n'a aucune façon d'avoir un vertex-cover avec seulement ces sommets verts.

$$M[c_i, B_i] = \begin{cases} \infty \text{ s'il existe } u, v \in B_i \text{ tel que } uv \in E(G) \text{ mais } u, v \notin c_i^{vert} \\ |c_i^{vert}| \text{ sinon} \end{cases}$$

Soit $B_i$ un noeud interne. On fait la même vérification, i.e. si une arête $uv \in E(G)$ est dans $B_i$ mais que $u$ et $v$ ne sont pas verts, on met $M[c_i, B_i] = \infty$. Sinon, on a 3 cas de figure.

- $B_i$ est un noeud de jointure. Soient $B_l$ et $B_r$ les enfants de $B_i$, avec $B_i = B_l = B_r$. On met

  $$M[c_i, B_i] = M[c_i, B_l] + M[c_i, B_r] - |c_i^{vert}|$$

  Pour expliquer un peu (ce qui serait optionnel en examen), il faut garder le même coloriage aux enfants $B_l$ et $B_r$ pour être compatible puisque $B_i = B_l = B_r$. Aussi, on peut librement prendre l'union des solutions minimum aux enfants car cette union couvrira les arêtes dans les sacs descendants de $B_l$ et $B_r$. On ne peut pas faire mieux que cette union car les arêtes sous $B_l$ et les arêtes sous $B_r$ sont disjointes.

- $B_i$ est un noeud d'introduction. Soit $B_j$ l'enfant de $B_i$ et $B_i = B_j \cup \{v\}$. Soit $c_{j^*}$ l'unique coloriage de $B_j$ compatible avec $c_i$. On met

  $$M[c_i, B_i] = M[c_{j^*}] + \begin{cases} 0 \text{ si } c_i(v) = rouge \\ 1 \text{ si } c_i(v) = vert \end{cases}$$

  Ceci fonctionne car on peut couvrir toutes les arêtes sous $B_j$ avec une solution au niveau $B_j$, et on suppose qu'on a déjà vérifié que les arêtes incidentes à $v$ dans $B_i$ sont couvertes.

- $B_i$ est un noeud d'oubli. Soit $B_j$ l'enfant de $B_i$ et $B_i = B_j \setminus \{v\}$. Il y a deux coloriages compatibles de $B_j$ selon la couleur de $v$, dénotons ces coloriages par $c_{j1}$ et $c_{j2}$. On met

  $$M[c_i, B_i] = \min(M[c_{j1}, B_j], M[c_{j2}, B_j])$$

  Ceci fonctionne car on peut couvrir toutes les arêtes dans $B_i$ ou un descendant en couvrant les arêtes dans $B_j$ ou un descendant.

La valeur finale à retourner est $\min_{c_r} M[c_r, B_r]$ où $B_r$ est la racine de $T$. Le nombre d'entrées totale à calculer est $O(2^{tw(G)} \cdot n)$. À chaque entrée, on doit vérifier que chaque arête $uv$ est couverte par les sommets verts, ce qui peut prendre un temps $O(tw(G)^2)$. La complexité de l'algorithme est donc $O(2^{tw(G)} \cdot tw(G)^2 \cdot n)$.

Un autre algorithme pour un vertex-cover minimum consiste à exécuter l'algorithme pour MAX-INDSET, obtenir la taille $x$ d'un ensemble indépendant maximum, et retourner $n-x$, qui est la taille minimum d'un vertex-cover. Mais c'est beaucoup moins amusant!

---

*Exercice* 6. Soit $G = (V, E)$ un graphe. Soit le problème suivant.

MIN-DOMSET
**Entrée :** Graphe $G = (V, E)$
**Paramètre :** $tw(G)$
**Sortie :** un ensemble dominant $X \subseteq V$ de taille minimum

Supposez qu'on vous donne un décomposition en arbre $T = (V_T, E_T)$ de $G$. Donnez un algorithme de programmation dynamique en temps $O(f(tw(G))n^c)$ sur $T$ pour résoudre le problème MIN-DOMSET. Il est suffisant de décrire vos récurrences.

Ici, il est commode d'utiliser trois couleurs pour vos sommets. Une couleur pour un sommet dans l'ensemble indépendant, une couleur pour un sommet qui n'y est pas, mais qui est présentement dominé, et une couleur pour un sommet qui n'y est pas, et qui n,est pas encore dominé.

---

**Solution.**

On suppose qu'on reçoit une jolie décomposition $T = (V_T, E_T)$. On utilise des coloriages $c_i : B_i \to \{vert, jaune, rouge\}$. L'interprétation est la suivante:
– *vert* veut dire "est dans l'ensemble dominant"
– *jaune* veut dire "n'est pas dans l'ensemble dominant, et n'est PAS encore dominé"
– *rouge* veut dire "n'est pas dans l'ensemble dominant, et est dominé".

On écrit $c^{vert}$, $c^{rouge}$ et $c^{jaune}$ pour dénoter les sommets verts, rouges et

jaunes, respectivement.

L'idée est que si un sommet $v$ n'est pas vert, il faut qu'il soit dominé. Par contre, peut-être que $v$ sera dominé par un sommet d'un sac que l'on n'a pas encore rencontré. Dans ce cas, $v$ est jaune. Lorsqu'on ajoute un sommet vert qui domine ce $v$, alors ce $v$ peut passer au rouge.

On définit $M[c_i, B_i]$ comme la taille minimum d'un ensemble dominant $D$ de $G[V_i] - c_i^{jaune}$ tel que $D \cap B_i = c_i^{vert}$ (ou $\infty$ si c'est impossible). Notez que les sommets jaunes n'ont pas encore besoin d'être dominés.

Soit $B_i$ une feuille. On met

$$M[c_i, B_i] = \begin{cases} \infty \text{ s'il y a } u \in c_i^{rouge} \text{ tel que } N(u) \cap c_i^{vert} = \emptyset \\ |c_i^{vert}| \text{ sinon} \end{cases}$$

Pourquoi? Puisqu'on a défini $c_i^{rouge}$ comme étant les sommets présentement dominés, on voit qu'il est impossible de dominer $u \in c_i^{rouge}$ si $u$ n'a pas de voisin vert présentement dans $B_i$.

Si $M[c_i, B_i]$ est un noeud interne, les situations impossibles dépendent du type de noeud. On a donc 3 cas de figure. Les cas sont plus complexes ici, car des sommets jaunes pourraient devenir rouges lorsque leur sommet dominant apparaît. On doit aussi s'assurer de ne pas laisser de sommet *jaune* ne jamais être dominé.

- $B_i$ est un noeud de jointure, avec enfants $B_l, B_r$ et $B_i = B_l = B_r$.

  Tout ce qui est vert ou jaune dans $B_i$ doit aussi l'être dans $B_l$ et $B_r$. Par contre, il est possible que $c_l(u) = jaune$ et $c_r(u) = rouge$ si $u$ a été dominé du côté $B_r$ précédemment, mais pas du côté $B_l$. En combinant les ensembles dominants correspondants, on s'assure que $u$ est dominé. On va dire que $c_l$ et $c_r$ sont compatibles si $c_l(u) = c_r(u) = c_i(u)$ lorsque $c_i(u) \in \{vert, jaune\}$, et lorsque $c_i(u) = rouge$, au moins un de $c_l(u)$ ou $c_r(u)$ est $rouge$, alors que l'autre est $jaune$ ou $rouge$. Alors

  $$M[c_i, B_i] = \min_{c_l, c_r \text{ compatibles}} (M[c_l, B_l] + M[c_r, B_r] - |c_i^{vert}|)$$

- $B_i$ est un noeud d'introduction, avec enfant $B_j$ et $B_i = B_j \cup \{v\}$. On doit diviser en trois sous-cas selon $c_i(v)$.

  - $c_i(v) = rouge$. Si $v$ n'a pas de voisin vert dans $B_i$, alors $M[c_i, B_i] = \infty$, car $v$ ne pourrait pas être dominé présentement (car $v$ n'a pas de voisin dans $B_j$ selon les propriétés des décompositions).

Si $v$ a un voisin vert dans $B_i$, alors soit $c_{j^*}$ le coloriage de $B_j$ tel que $c_{j^*}(u) = c_i(u)$ pour tout $u \in B_j$. Alors les relations de domination ne changent pas par rapport à $B_j$, ni le nombre de sommets dominants, et donc

$$M[c_i, B_i] = M[c_{j^*}, B_j]$$

- $c_i(v) = jaune$. Si $v$ a un voisin vert dans $B_i$, alors $v$ devrait plutôt être rouge. Donc si $v$ a un voisin vert dans $B_i$, on met $M[c_i, B_i] = \infty$. Sinon, la solution ne change pas par rapport à $B_j$, et donc

$$M[c_i, B_i] = M[c_{j^*}, B_j]$$

- $c_i(v) = vert$. En ajoutant $v$ dans l'ensemble dominant, les voisins jaunes de $v$ deviennent rouges car ils sont maintenant dominés. Soit $C$ l'ensemble des coloriages de $B_j$ tels que pour tout $c_j \in C$, $c_i(u) = c_j(u)$ si $u \notin N(v) \cap B_j$ ou si $c_i(u) \in \{vert, jaune\}$, et tel que $c_j(u) \in \{jaune, rouge\}$ si $u \in N(v)$ et $c_i(u) = rouge$. Alors

$$M[c_i, B_i] = 1 + \min_{c_j \in C} M[c_j, B_j]$$

(pensez-y!)

- $B_i$ est un noeud d'oubli, avec enfant $B_j$ et $B_i = B_j \setminus \{v\}$. Le seul danger est que $v$ soit jaune et n'ait pas été dominé. Sinon, on peut reprendre n'importe quel ensemble dominant qui contient $v$ (vert) ou tel que $v$ est dominé (rouge). Soit $c_{j1}$ le coloriage de $B_j$ avec les mêmes couleurs que $c_i$ et $c_i(v) = vert$, et $c_{j2}$ tel que $c_i(v) = rouge$. Alors

$$M[c_i, B_i] = \min(M[c_{j1}, B_j], M[c_{j2}, B_j])$$

Facile non? Le nombre d'entrées à calculer est $O(3^{tw(G)} \cdot n)$. Le pire cas pour calculer une entrée est à un noeud de jointure, où il faut considérer $O(3^{tw(G)})$ coloriages chez les deux enfants et faire la somme. Le calcul d'une entrée peut donc prendre un temps jusqu'à $O(3^{tw(G)} \cdot 3^{tw(G)}) = O(3^{2tw(G)})$. La complexité totale est donc le nombre d'entrées à calculer fois le temps par entrée, ce qui est $O(3^{tw(G)} \cdot n \cdot 3^{2tw(G)}) = O(3^{3tw(G)} \cdot n)$.

Notez qu'avec un peu de raffinement, il est possible d'atteindre $O(3^{tw(G)} \cdot n)$.

Bien sûr, il faudrait une preuve détaillée pour montrer que ces récurrences fonctionnent. Nous allons nous en passer ici. On note que la valeur finale à

retourner est la valeur maximum de $M[c_r, B_r]$ avec $B_r$ la racine, où on regarde seulement les $c_r$ qui n'ont pas de sommet jaune.

---

*Exercice* 7. Soit $G = (V, E)$ un graphe *orienté* et sans cycle de taille 2. La *version non-orientée* de $G$ est le graphe $G'$ obtenu en ignorant la direction des arêtes. Une décomposition en arbre d'un graphe orienté $G$ réfère à une décomposition de la version non-orientée de $G$.

Dans le problème FEEDBACK ARC SET, on veut ordonner les sommets de façon à avoir un nombre minimum d'arêtes qui vont "vers l'arrière".

FEEDBACK ARC SET
**Entrée :** Graphe orienté $G = (V, E)$
**Paramètre :** $tw(G')$, où $G$ est la version non-orientée de $G$
**Sortie :** une permutation $(v_1, v_2, \ldots, v_n)$ telle que le nombre d'arêtes $(v_j, v_i)$ avec $j > i$ est minimum.

Supposez qu'on vous donne un décomposition en arbre $T = (V_T, E_T)$ de la version non-orientée de $G$. Donnez un algorithme de programmation dynamique en temps $O(f(tw(G))n^c)$ sur $T$ pour résoudre ce problème. Il est suffisant de décrire vos récurrences.

Ici, il ne faut pas considérer les sous-ensembles à chaque sac, mais plutôt les permutations à chaque sac.

---

**Solution.**

Le technique de coloriage ne fonctionne pas ici. À la place, à chaque $B_i$, on va tester toutes les permutations de $B_i$. On dénote par $p_i$ une permutation des éléments de $B_i$ et on écrit $u \prec_{p_i} v$ si $u$ vient avant $v$ dans $p_i$. Deux permutations $p_i$ de $B_i$ et $p_j$ de $B_j$ sont compatibles si elles s'entendent sur l'ordre des éléments communs, c'est-à-dire si, pour tout $u, v \in B_i \cap B_j$, on a $u \prec_{p_i} v \Leftrightarrow u \prec_{p_j} v$.

On définit $M[p_i, B_i]$ comme le nombre minimum d'arêtes allant vers l'arrière d'une permutation de $V_i$.

Si $B_i$ est une feuille, alors

$$M[p_i, B_i] = |\{(u, v) \in E(G) : v \prec_{p_i} u\}|$$

c'est-à-dire, simplement le nombre d'arêtes allant vers l'arrière selon $p_i$.

Si $B_i$ est un noeud interne, on a trois cas de figure.

- $B_i$ est un noeud de jointure avec enfants $B_l, B_r$ et $B_i = B_l = B_r$. On a simplement

$$M[p_i, B_i] = M[p_i, B_l] + M[p_i, B_r] - |\{(u, v) \in E(G) : v \prec_{p_i} u\}|$$

  Ceci correspond à combiner deux sous-permutations de $V_l$ au niveau $B_l$ et de $V_r$ au niveau $B_r$. Ça fonctionne parce que si ces deux permutations s'entendent sur l'ordre dans $B_i$, il y aura toujours moyen de placer les éléments de $V_l \setminus B_i$ et $V_r \setminus B_r$ dans le bon ordre. Il n'y a pas d'arête arrière à considérer parce qu'il n'y a pas d'arête entre $V_l \setminus B_i$ et $V_r \setminus B_r$.

- $B_i$ est un noeud d'introduction avec enfant $B_j$ et $B_i = B_j \cup \{v\}$. Soit $p_{j^*}$ le seul ordre de $B_j$ compatible avec $B_i$. On a simplement

$$\begin{aligned} M[p_i, B_i] =& M[c_{j^*}, B_j] + \\ & |\{(u, v) \in E(G) : v \prec_{p_i} u\}| + \\ & |\{(v, u) \in E(G) : u \prec_{p_i} v\}| \end{aligned}$$

  Ceci fonctionne car ça correspond à prendre le seul ordre compatible pour $V_j$, et d'ajouter les arêtes arrières touchant $v$.

- $B_i$ est un noeud d'oubli avec enfant $B_j$ et $B_i = B_j \setminus \{v\}$. On a

$$M[p_i, B_i] = \min_{p_j}(M[p_j, B_j])$$

  où la minimisation itère sur les $p_j$ compatibles avec $p_i$. Notez qu'il y en a $O(tw(G))$, car un $p_j$ compatible est obtenu en insérant $v$ à un des $tw(G)$ endroits de $p_i$. Ensuite, on peut simplement reprendre n'importe quelle solution pour $V_l$.

Le nombre d'entrées à calculer est $O((tw(G) + 1)! \cdot n)$, où le $(tw(G) + 1)!$ correspond au nombre de permutations d'un sac, qui sont de taille au plus $tw(G) + 1$. Le pire cas est le calcul d'un noeud d'oubli. Il y a $O(tw(G))$ permutations $p_j$ compatibles à tester, chacune en temps $O(tw(G))$ pour construire la permutation, donc en temps total $O((tw(G))^2)$. Pour les autres types de noeuds, on peut aussi les calculer en temps $O((tw(G))^2)$, le temps requis étant dominé par le calcul de nombre d'arêtes traversantes. La complexité totale est donc $O((tw(G) + 1)! \cdot n \cdot (tw(G))^2)$.

## Exercice récapitulatifs

*Exercice* 8. Dans le problème SEQUENCES-INTRUES, on reçoit $m$ séquences de caractères $S = \{S_1, \ldots, S_m\}$ de même longueur. On veut savoir si on peut supprimer $k$ de nos séquences de façon à ce que chaque paire de séquences soit à distance au plus $d$, où $d$ nous est donné.

C'est-à-dire, est-ce qu'il existe $S' \subseteq S$ tel que $|S'| \leq k$, et tel que pour tout $S_i, S_j \in S \setminus S'$, on a $d(S_i, S_j) \leq d$?

Montrez que cet algorithme est FPT en le paramètre $k$, le nombre de séquences à supprimer. Notez que $d$ ne fait pas partie des paramètres.

---

**Solution.**

Ce problème n'est rien d'autre que VERTEX-COVER en déguisement.

Pourquoi? On va dire que $S_i$ et $S_j$ sont en conflit si $d(S_i, S_j) > 2$. Soit $G = (V, E)$ le graphe où $V = \{S_1, \ldots, S_m\}$ et $S_iS_j \in E(G)$ si $S_i$ et $S_j$ sont en conflit.

Pour que toutes les séquences restantes soient à distance au plus $d$, il faut returer au moins une séquence par conflit. Donc pour chaque $S_iS_j \in E(G)$, il faut enlever $S_i$ ou $S_j$ (ou les deux). On veut donc savoir s'il existe au plus $k$ séquences qui couvrent chacune des arêtes (car une fois qu'on retire ces $k$ séquences, il n'y a plus de conflit et tout ce qui reste est à distance au plus $d$).

Ceci est exactement VERTEX-COVER. On peut donc utiliser l'algorithme $O(2^k(n + m))$, ou tout autre algorithme FPT.

---

*Exercice* 9. Soit $P$ un problème de minimisation résoluble en temps polynomial (et ce indépendamment d'un paramètre). Montrez que $P$ admet un noyau de taille $O(1)$ selon le paramètre $k$, la valeur d'une solution minimum.

---

**Solution.**

Soit $(I, k)$ une instance paramétrée. Notre algorithme de kernelisation résout $I$ et y trouve la solution optimale $OPT$ (indépendamment de $k$). Si $OPT \leq k$, l'algorithme retourne $true$, ce qui est de taille $O(1)$ (techniquement, on pourrait retourner un sommet individuel, ou toute instance $I'$ sur laquelle la taille de la solution est par exemple 0, ce qui constituerait un noyau constant sur lequel la réponse est toujours $true$). Si $OPT > k$, on retourne $null$, ce qui est de taille $O(1)$. De cette façon, $(I, k)$ est une instance positive si et seulement si ce que l'algorithme retourne est une instance positive.

---

*Exercice* 10. Dans MIN-ONES-2-SAT-FACILE, on reçoit des clauses $C_1, \ldots, C_m$ chacune avec deux variables. On suppose également que chaque clause a ses deux variables positives. Ceci rend la question beaucoup plus facile que si on permettait des variables négatives. Le but est de savoir s'il existe une assignation qui satisfait les clauses où au plus $k$ variables sont $true$, et les autres à $false$.

Montrez qu'on peut obtenir un noyau pour MIN-ONES-2-SAT-FACILE, où le paramètre $k$ est le nombre maximum de variables à $true$.

---

**Solution.**

En fait ce problème est aussi équivalent à vertex-cover. Il faut voir les sommets comme les variables et les arêtes comme des clauses. Nous décrivons le même noyau qu'en classe, mais avec la terminologie de satisfaisabilité.

On a une première règle de réduction triviale.

**Règle 1**. Si une variable $x_i$ n'apparaît dans aucune clause, retirer $x_i$ de l'instance.

**Règle 2**. Pour une variable $x_i$, soient $(x_i \vee x_{j_1}), \ldots, (x_i \vee x_{j_l})$ les clauses où $x_i$ apparaît. Si $l > k$, on met $x_i = true$, on décrémente $k$ de 1 et on élimine les clauses satisfaites.

Pourquoi la règle 2 est-elle saine? Si $l > k$ et qu'on met $x_i = false$, alors on sera obligé de mettre $x_{j_1} = true, \ldots, x_{j_l} = true$. Mais quand $l > k$, on dépasse le nombre voulu de variables vraies.

On peut donc supposer que chaque variable $x_i$ apparaît dans au plus $k$ clauses. Donc $x_i$ peut satisfaire au plus $k$ clauses en mettant $x_i = true$. Puisqu'on a droit à $k$ variables seulement, s'il y a plus de $k^2$ clauses, on peut immédiatement retourner $null$.

On peut donc supposer qu'il y a au plus $k^2$ clauses. Puisque chaque variable est dans au moins une clause (règle 1), il y a au plus $2k^2$ clauses.

---

*Exercice* 11. Donnez un algorithme de branchement FPT pour le problème MIN-ONES-2-SAT ci-haut, mais sans supposer que chaque clause a toujours deux variables positives.

---

**Solution.**

Si on pouvait trouver une clause $C = (x_i \vee x_j)$ avec ses deux variables positives, on pourrait brancher sur $C$.

Que se passe-t-il s'il n'y a pas de telle clause? Alors on peut trivialement satisfaire toutes les clauses en mettant chaque variable à $false$.

L'idée est donc la suivante:

– s'il n'y a pas de clause $C = (x_i \vee x_j)$ avec deux variables positives, retourner l'assignation avec tout à $false$.

– sinon, soit $C = (x_i \vee x_j)$. Brancher sur $x_i = true$, et sur $x_j = true$. Dans les deux cas, réduire $k$ de 1.

Le cas terminal est si $k < 0$ (retourner $null$) ou si $k \geq 0$ et le nombre de clauses est 0 (retourner $true$, ou encore l'assignation courante).

Cet algorithme prendra un temps $O(2^k n^c)$. Nous vous laissons le soin d'écrire le pseudo-code détaillé.

---

*Exercice* 12. Rappelez-vous du problème LIVRAISON-CIRCULAIRE de la section 7.2.2 des notes de cours. On a un cycle $G = (V, E)$ avec $n$ sommets et on a un ensemble de paquets représentés par des couples $(s_i, t_i) \in V \times V$, où $s_i$ est le départ et $t_i$ l'arrivée. Il faut choisir une direction pour chaque paquet de façon à ce que la charge maximum d'une arête soit au plus $k$. La charge d'une arête est le nombre de chemins qui passent par cette arête.

Montrez que LIVRAISON-CIRCULAIRE est FPT en le paramètre $n + k$.

*Note :* je ne sais pas si c'est FPT en le paramètre $k$ seulement. Si vous trouvez, dites-le moi!

---

**Solution.**

Soient $(s_1, t_1), \ldots, (s_m, t_m)$ la liste des $m$ paquets à livrer. La seule chose qui nous empâche de faire une force brute est que $m$ ne fait pas partie des paramètres.

Par contre, on remarque la chose suivante. Soit $s_i \in V(G)$ et soit $D_i$ l'ensemble des paquets qui ont comme point de départ le sommet $s_i$. Si $|D_i| > 2k$, alors nécessairement une des deux arêtes incidentes à $s_i$ aura une charge supérieure à $k$ (assurez-vous de savoir pourquoi!). Si c'est le cas, on retourne *null*. Sinon, pour tout $s_i \in V(G)$, on a $|D_i| \leq 2k$. Le nombre de paquets total à livrer est donc borné par $n \cdot 2k$.

On peut maintenant appliquer la force brute. Pour chaque paquet, on a deux choix de direction. Il y a donc $2^m$ possibilités de combinaisons de choix de livraison. On les essaie toutes et on regarde si un de ces choix donne une charge moins grande que $k$. Puisque $2^m \leq 2^{n \cdot 2k}$, la complexité totale d'un tel algorithme sera $O(2^{n \cdot 2k} \cdot n^c)$, ce qui est FPT en le paramètre $n + k$. (notez en fait qu'en ayant $n \cdot 2k$ paquets, on a obtenu un noyau selon nos paramètres)

---

*Exercice* 13. Considérez la variante suivante de LIVRAISON-CIRCULAIRE. On a la même entrée qu'au problème précédent, mais on veut choisir un chemin pour chaque paquet en minimisant le nombre d'arêtes de charge 2 ou plus. Montrez que ce problème est résoluble en temps polynomial.

*Note* : cet exercice n'est pas un problème FPT, mais il entraînera tout de même votre "mode de réflexion algorithmique".

Si vous aimez les défis, une version paramétrée serait la suivante : décider s'il est possible d'avoir au plus $k$ arêtes de charge $c$ ou plus, avec $k + c$ comme paramètre. Je ne sais pas si c'est FPT.

---

**Solution.**

Soient $(s_1, t_1), \ldots, (s_m, t_m)$ la liste des $m$ paquets à livrer. On va dire qu'une arête $uv$ est saturée si deux chemins ou plus passent par $uv$. Fixons une arête $uv \in E$ et demandons-nous quel est l'optimal si on ne veut pas saturer $uv$.

En fait, on remarque que pour chaque $uv \in E$, il n'y a que $m + 1$ solutions possibles qui ne saturent pas $uv$. Une première façon est de n'avoir aucun paquet qui passe par $uv$. Cette solution est unique car pour chaque paquet, il n'y a qu'une direction possible qui évite $uv$. Puisque chaque choix est forcé, le nombre d'arêtes saturées d'une telle solution est donc facile à trouver en temps $O(mn)$ ou mieux (en exercice).

Une autre façon de ne pas saturer $uv$ est d'avoir exactement un paquet $(s_i, t_i)$ qui passe par $uv$, et il y a $m$ choix de paquet. Fixons un paquet $(s_i, t_i)$ et choisissons l'unique direction de ce paquet qui passe par $uv$. Pour éviter de saturer $uv$, tous les autres paquets doivent éviter $uv$ et leur choix de chemin est encore une fois forcé. On peut donc calculer le nombre d'arêtes saturées d'une solution qui ne sature pas $uv$ et dans laquelle seul $(s_i, t_i)$ passe par $uv$ facilement. Ceci prendra également un temps $O(mn)$.

Pour trouver l'optimal, il suffit d'essayer toutes les possibilités. Pour chaque $uv \in E$, on teste toutes les $m+1$ façons de ne pas saturer $uv$. On garde la solution qui minimise le nombre d'arêtes saturées. Il y a $|E(G)| = n$ arêtes à tester, et pour chacune on essaie $m + 1 \in O(m)$ solutions. Chaque solution se calcule en temps $O(mn)$, et donc la complexité totale est $O(nm \cdot mn) = O(n^2m^2)$.

---