I Arbres couvrants

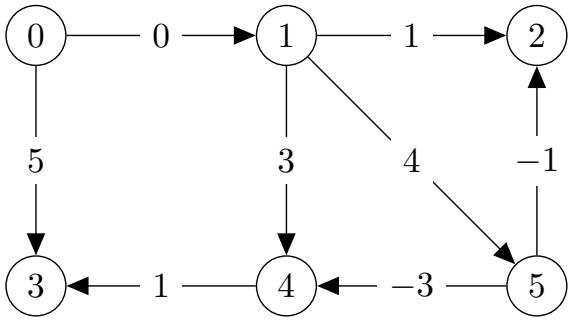
Soit $G = (V, E)$ un graphe pondéré.

1. Soit C un cycle de G et $e = \{u, v\}$ une arête de C dont le poids est strictement supérieur au poids des autres arêtes de C. Montrer que e ne peut pas appartenir à un arbre couvrant de poids minimum de G.
2. (Propriété d'échange) Soient T_1, T_2 deux arbres couvrants de G et e_1 une arête de $T_1 - T_2$. Montrer qu'il existe une arête e_2 de T_2 telle que $T_1 - e_1 + e_2$ (le graphe obtenu en remplaçant e_1 par e_2 dans T_1) est un arbre couvrant de G.
3. Le nombre de domination $d(G)$ d'un graphe $G = (V, E)$ est le cardinal minimum d'un ensemble $S \subseteq V$ tel que $\forall v \in V$, $v \in S$ ou v est adjacent à un sommet de S.
   Montrer que si G est connexe alors $d(G) \leq \dfrac{|V|}{2}$.
4. Soit G un graphe connexe et T un arbre couvrant de poids minimum. La **largeur** d'un chemin C est le poids maximum d'une arête de C. Soient u et v deux sommets de G. Montrer que le chemin de u à v dans T est de largeur minimum, c'est-à-dire qu'il n'existe pas d'autre chemin de u à v de plus petite largeur.
5. Montrer que si tous les poids des arêtes de G sont différents, alors G admet un unique arbre couvrant de poids minimum.
6. Soit T_1 un arbre couvrant de poids minimum de G et T_2 le 2ème plus petit arbre couvrant, c'est-à-dire l'arbre couvrant de poids minimum en excluant T_1. Montrer que T_1 et T_2 diffèrent d'une arête et en déduire un algorithme pour trouver T_2.

II Tri topologique

On rappelle qu'un **ordre topologique** d'un graphe orienté $G = (V, E)$ est une liste $v_1, ..., v_n$ des sommets de G telle que $(v_i, v_j) \in G \implies i < j$.

1. Donner un ordre topologique pour le graphe ci-dessous.



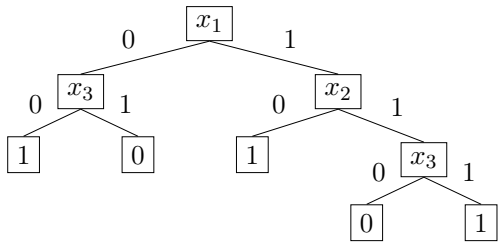
2. Dans quel algorithme du cours utilise t-on un ordre topologique ?
3. Montrer qu'un parcours en profondeur, sur un graphe acyclique, énumérant les sommets par ordre décroissant de fin de visite est un ordre topologique.
4. En déduire une fonction `tri : int list array -> int list` renvoyant un tri topologique d'un graphe représenté par liste d'adjacence.
5. Expliquer comment trouver les plus courts chemins depuis un sommet d'un graphe orienté pondéré acyclique en complexité linéaire.
6. En déduire une fonction `distances g s` renvoyant la liste des distances des sommets du graphe g depuis le sommet s, en complexité linéaire.
7. Comment adapter `distances` pour obtenir les plus courts chemins, au lieu des distances ?
8. Comment adapter `distances` pour obtenir les plus longs chemins, au lieu des distances ?
9. Comparer avec les algorithmes vus en cours pour trouver les plus courts chemins.

A4 – Arbres de décision

On considère un ensemble de variables propositionnelles $\mathcal{X} = \{x_1, \ldots, x_n\}$ muni de l'ordre total où $x_i < x_j$ si et seulement si $i < j$. Un *arbre de décision* sur $\mathcal{X}$ est un arbre binaire dont les feuilles sont étiquetées par 0 ou par 1, et dont les nœuds internes sont étiquetés par une variable de $\mathcal{X}$ et ont deux enfants appelés *enfant* 0 et *enfant* 1. On impose que si un nœud interne n étiqueté par x_i a un descendant n' qui est un nœud interne étiqueté par x_j alors $x_i < x_j$.

Un arbre de décision T sur $\mathcal{X}$ décrit une fonction booléenne Φ_T qui à toute *valuation* $\nu : \mathcal{X} \to \{0, 1\}$ associe une valeur de vérité calculée comme suit : si T consiste exclusivement d'une feuille étiquetée $b \in \{0, 1\}$ alors la fonction Φ_T s'évalue à b quelle que soit ν. Sinon, on considère le nœud racine n de T et la variable x_i qui l'étiquette, on regarde la valeur $\nu(x_i) \in \{0, 1\}$ que ν donne à x_i, et le résultat de l'évaluation de Φ_T sous ν est celui de l'évaluation de $\Phi_{T'}$ sous ν, où T' est le sous-arbre de T enraciné en l'enfant b de n.

Question 0. On considère l'arbre de décision T_0 suivant et la fonction Φ_{T_0} qu'il définit. Évaluer cette fonction pour la valuation donnant à x_1, x_2, x_3 respectivement les valeurs $0, 1, 0$. Donner un exemple de valuation sous laquelle cette formule s'évalue en 0.



Question 1. On considère la formule de la logique propositionnelle $(x_1 \wedge x_2) \vee \neg(x_1 \wedge \neg x_3)$. Construire un arbre de décision sur les variables $x_1 < x_2 < x_3$ qui représente la même fonction.

Question 2. Quels arbres de décision représentent des tautologies ? des fonctions satisfiables ?

Question 3. Étant donné un arbre de décision représentant une fonction Φ, expliquer comment construire un arbre de décision représentant sa négation $\neg\Phi$.

Question 4. Étant donné un arbre de décision représentant une fonction Φ, donner le pseudocode d'un algorithme qui calcule une représentation de Φ sous la forme d'une formule de la logique propositionnelle. Analyser sa complexité en temps et en espace.

Question 5. Étant donné deux arbres de décision représentant des formules Φ_1 et Φ_2 sur le même ensemble de variables et avec le même ordre, expliquer comment construire un arbre de décision représentant $\Phi_1 \wedge \Phi_2$. Quelle est sa complexité en temps et la taille de l'arbre ainsi obtenu ?

Question 6. Étant donné un arbre de décision et la séquence de variables $x_1, \ldots, x_n$, donner le pseudocode d'un algorithme qui calcule combien de valuations satisfont la fonction booléenne qu'il capture. Quelle est sa complexité en temps ?

Question 7. Peut-on efficacement récrire une formule quelconque de la logique propositionnelle en arbre de décision ?

Suite des questions

On se propose dans les quelques prochaines questions de démontrer formellement l'intuition de la question 7.

Question 7a. Une formule booléenne en *forme normale disjonctive*, également appelée DNF, est une formule qui est une disjonction de conjonctions de littéraux (c'est-à-dire une variable ou sa négation).

Montrer qu'étant donné un arbre de décision on peut efficacement calculer une formule en DNF qui représente la même fonction.

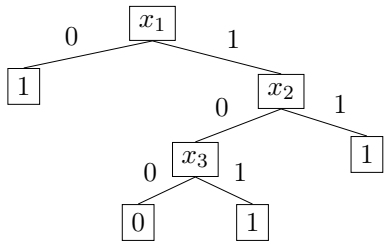
Question 7b. Montrer que, pour tout n, la formule $\bigwedge_{1 \leq i \leq n}(x_i \vee y_i)$ n'admet pas de représentation concise sous la forme d'une DNF.

Question 7c. Conclure.

Corrigé

Question 0. L'évaluation renvoie 1 (pour la feuille tout à gauche). Pour une valuation qui renvoie 0, on peut par exemple envoyer x_1, x_2, x_3 à 1, 1, 0 respectivement.

Question 1.



Question 2. Un arbre de décision représente une tautologie si et seulement si toutes ses feuilles sont étiquetées par 1. En effet, dans ce cas il est clair que la fonction représentée renvoie toujours 1. À l'inverse, par contraposition, s'il y a une feuille étiquetée par 0 alors en suivant un chemin de la racine à cette feuille on obtient une valuation partielle qu'on peut compléter en une valuation totale justifiant que la fonction n'est pas une tautologie. Noter que dans ce cas on pourrait représenter la même fonction de manière plus concise par un arbre n'ayant qu'une feuille.

De la même manière, un arbre de décision représente une fonction satisfiable si et seulement s'il a une feuille étiquetée par 1.

Question 3. Il suffit d'inverser l'étiquette de chaque feuille (c'est-à-dire remplacer 0 par 1 et 1 par 0). Pour montrer que c'est correct, si l'on considère une valuation ν, alors l'évaluation de l'arbre original T et de l'arbre récrit T' sous cette valuation mène dans T' à la feuille correspondant à la feuille atteinte dans T, et ainsi on renvoie 0 (resp., 1) dans T' si et seulement si on renvoie 1 (resp., 0) dans T. Ainsi, T' capture bien $\neg\Phi$.

Question 4. *[On peut discuter comment on représente concrètement la formule propositionnelle ? La représentation comme un arbre est mentionnée dans le programme, sinon on peut aussi la représenter comme une chaîne de caractères avec des parenthèses.]*

Possibilité 1 : récriture vers des décisions. On récrit chaque feuille en vrai ou faux, et chaque nœud interne en une disjonction qui choisit quel sous-arbre utiliser suivant la valeur de la variable testée.

```
Entrée : arbre T

Def Récrire(n):
  Si n est une feuille:
    Renvoyer son étiquette
  Fin si
  x <- Variable testée en n
  n0, n1 <- Enfants de n
  F0 <- Récrire(n0)
  F1 <- Récrire(n1)
  Renvoyer Or(And(x, F0), And(Not(x), F1))
Fin def

Renvoyer Récrire(racine de T)
```

La complexité de cet algorithme est manifestement linéaire en temps et en espace (à condition que les arbres soient passés par référence).

Possibilité 2 : récriture vers une forme normale disjonctive (DNF). On récrit en une disjonction de clauses conjonctives correspondant à l'étiquette des chemins vers des feuilles 1.

```
Entrée : arbre T

Def Récrire(nœud n, liste de littéraux L)
  Si n est une feuille:
    Si elle est étiquetée 1:
      Renvoyer la liste singleton ["AND".join(L)]
    Sinon:
      Renvoyer la liste vide
    Fin si
  Fin si
  x <- Variable testée en n
  n0, n1 <- Enfants de n
  Renvoyer (Récrire(n0, (-x)::L) union Récrire(n1, (+x)::L))
Fin def

Renvoyer "OR".join(Récrire(racine de T, []))
```

La complexité de cet algorithme est quadratique en temps et en espace (la borne supérieure est claire, la borne inférieure s'obtient en considérant un long chemin $x_1, \ldots, x_n$ de taille n suivi d'un arbre binaire complet à n feuilles et $n-1$ nœuds internes sur les variables suivantes : on a un arbre de taille $O(n)$ mais la DNF a taille $O(n^2)$ vu que chaque feuille lui fait répéter le préfixe $x_1, \ldots, x_n$.

Question 5. On définit la transformation récursivement comme suit (un peu comme le produit de deux automates) :

```
Entrée : deux arbres de décision Ta et Tb

infty := |Ta| + |Tb| + 1

Def Conjonction(nœud na de Ta, nœud nb de Tb):
  Si na et nb sont des feuilles:
```

```
    ba <- étiquette de na
    bb <- étiquette de nb
    Renvoyer une feuille étiquetée par (ba ET bb)
  Fin si
  xa <- variable de na (ou infty si na est une feuille)
  xb <- variable de nb (ou infty si nb est une feuille)
  Si xa == xb:
    na0, na1 <- enfants de na
    nb0, nb1 <- enfants de nb
    T0 <- Conjonction(na0, nb0)
    T1 <- Conjonction(na1, nb1)
    Renvoyer un nœud interne étiqueté par x et ayant T0 et T1 comme enfants
  Fin si
  // na et nb ont des variables différentes
  Si xa < xb:
    na0, na1 <- enfants de na
    T0 <- Conjonction(na0, nb)
    T1 <- Conjonction(na1, nb)
    Renvoyer un nœud interne étiqueté par xa et ayant T0 et T1 comme enfants
  Sinon:
    nb0, nb1 <- enfants de nb
    T0 <- Conjonction(na, nb0)
    T1 <- Conjonction(na, nb1)
    Renvoyer un nœud interne étiqueté par xb et ayant T0 et T1 comme enfants
  Fin si
Fin def

Renvoyer Conjonction(racine de Ta, racine de Tb)
```

Justification de la correction : pour toute valuation, le chemin de la racine du nouvel arbre à une feuille franchit une succession de nœuds internes correspondant à des paires (na, nb) dans la construction de l'algorithme, et la suite des premières et secondes valeurs donnent les chemins compatibles avec cette valuation dans Ta et Tb. Or, la feuille à laquelle on arrive dans l'arbre récrit porte pour étiquette le ET des étiquettes des deux feuilles auxquelles on arrive dans les arbres originaux, donc on obtient bien le bon résultat.

Pour prouver la terminaison de cet algorithme, il faut noter l'invariant que Conjonction est toujours appelé avec une paire de nœuds qui sont descendants de chacun des deux nœuds précédents, avec au moins une des deux relations qui est stricte. Ainsi, l'algorithme termine, et le nombre d'appels est au plus en taille de Ta fois taille de Tb. Cette borne inférieure ne peut pas être améliorée en général : si on considère un arbre sur des variables $x_1, \ldots, x_k$ et un autre sur des variables $x_{k+1}, \ldots, x_{k'}$, l'effet de l'algorithme sera de créer une copie du deuxième arbre pour remplacer chacune des feuilles étiquetées 1 du premier arbre, or le nombre de feuilles d'un arbre binaire complet (c'est-à-dire où chaque nœud interne a deux enfants) est linéaire en sa taille.

Question 6. *Réponse naïve possible en premier lieu : on peut tester toutes les valuations. Dans ce cas, on s'attend à ce que le candidat remarque que la complexité est mauvaise.*

Il n'est pas compliqué de compter efficacement les valuations acceptantes, mais il faut faire attention aux variables manquantes dans l'arbre.

```
Entrée : un arbre de décision T, le nombre k de variables
  (on identifie les variables avec x1... xk)
```

```
Sortie : nombre de valuations satisfaisantes de T

Def Compte(nœud n, indice de la variable précédente xprev):
  Si n est une feuille étiquetée 0:
    Renvoyer 0
  Fin si
  Si n est une feuille étiquetée 1:
    Renvoyer 2^{k-iprev)
  Fin si
  // n est un nœud interne
  i <- indice de la variable testée en n
  n0, n1 <- enfants de n
  factor <- 2^{i-iprev-1} // valuations de variables précédemment sautées
  v0 <- Compte(n0, i)
  v1 <- Compte(n1, i)
  Renvoyer factor*(v0 + v1)
Fin def

Renvoyer Compte(racine de T, 0)
```

La complexité de l'algorithme est manifestement linéaire (pour des opérations arithmétiques en temps constant ; elle est polynomiale sinon).

Question 7. On ne s'attend pas à ce qu'une récriture efficace soit possible, parce qu'étant donné un arbre de décision, on peut facilement déterminer si la fonction correspondante est satisfiable (par la question 2, ou la question 6) ; or on sait qu'il est "difficile" de déterminer si une formule booléenne est satisfiable. (Cette notion est au programme de l'option informatique, même si la notion de NP-complétude n'est bien sûr pas exigible.)

Question 7a. C'est la possibilité 2 de la question 4.

Question 7b. On montre qu'une représentation de cette fonction sous forme de DNF doit avoir au moins 2^n clauses. Pour une valuation qui satisfait la fonction, on appelle *clause témoin* une clause qui est rendue vraie par la valuation : chaque valuation satisfaisante a au moins une clause témoin.

À présent, considérons les valuations satisfaisantes $\nu : \{x_1, \ldots, x_n, y_1, \ldots, y_n\} \to \{0, 1\}$ de la formule qui sont minimales, c'est-à-dire celles où, pour chaque $1 \leq i \leq n$, précisément l'un de x_i, y_i est assigné à vrai. Il y a clairement 2^n telles valuations.

À présent, supposons par l'absurde que la DNF a strictement moins de 2^n clauses. Par le principe des tiroirs, ceci implique que deux valuations minimales ν et ν' ont la même clause témoin. Soit z_i une variable telle que $\nu(z_i) \neq \nu'(z_i)$, ce qui existe forcément car les valuations minimales sont différentes. Sans perte de généralité, on va supposer que ν rend x_i vrai (et donc y_i faux par minimalité) et que ν' rend x_i faux (et donc y_i vrai car c'est une valuation satisfaisante). La clause comporte forcément l'un des littéraux x_i, y_i, puisque sinon elle est satisfaite par une valuation non satisfaisante qui rend x_i et y_i tous deux faux. Mais ce littéral ne peut être ni x_i (sinon ν' ne satisfait pas la clause), ni y_i (sinon ν ne la satisfait pas), on a donc une contradiction.

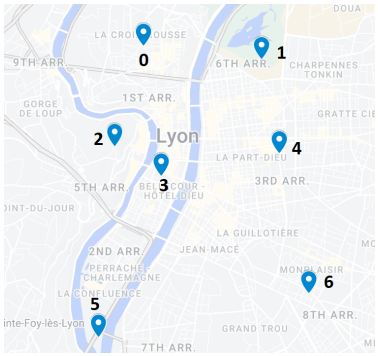
On a donc démontré par l'absurde que toute représentation de la fonction comme DNF doit avoir au moins 2^n clauses.

Question 7c. Posons $n \in \mathbb{N}$ et considérons la fonction définie à la question 7b sur $2n$ variables. Si cette fonction a une représentation sous la forme d'un arbre de décision de taille k, alors on sait par la

III Algorithme de Christofides

On considère un graphe $G = (V, E)$ **non orienté**, **complet** et **pondéré** par une fonction $w : E \longrightarrow \mathbb{R}^+$. On supposera dans toute la suite que $V = \{0, ..., n-1\}$ où n est le nombre de sommets. Si G' est un graphe ou un ensemble d'arêtes, son **poids** $w(G')$ est la somme des poids des arêtes de G'.

On utilisera comme exemple le graphe G_{ex} suivant, représenté par matrice d'adjacence pondérée, dont les sommets sont des points d'intérêts de Lyon (numérotés de 0 à 6) et chaque arête correspond à une distance euclidienne entre les deux points :



$$G_{ex} = \begin{pmatrix} 0.0 & 2.3 & 1.75 & 2.2 & 2.9 & 4.8 & 5.1 \\ 2.3 & 0.0 & 2.9 & 2.6 & 1.5 & 5.4 & 4.0 \\ 1.75 & 2.9 & 0.0 & 0.9 & 2.9 & 3.2 & 4.2 \\ 2.2 & 2.6 & 0.9 & 0.0 & 2.1 & 2.9 & 3.2 \\ 2.9 & 1.5 & 2.9 & 2.1 & 0.0 & 4.4 & 2.5 \\ 4.8 & 5.4 & 3.2 & 2.9 & 4.4 & 0.0 & 3.7 \\ 5.1 & 4.0 & 4.2 & 3.2 & 2.5 & 3.7 & 0.0 \end{pmatrix}$$

On suppose que G est **métrique**, c'est-à-dire que w vérifie l'inégalité triangulaire : $\forall x, y, z \in V, \quad w(x, y) \leq w(x, z) + w(z, y)$

Un **cycle hamiltonien** dans un graphe est un cycle qui passe exactement une fois par chaque sommet. Dans la suite, on représente un cycle hamiltonien par sa liste de sommets, dans l'ordre (contenant donc chaque sommet exactement une fois).

Le **problème du voyageur de commerce (TSP)** consiste à trouver un cycle hamiltonien de poids minimum.

L'**algorithme de Christofides** donne une $\frac{3}{2}$-approximation du TSP. Voici son fonctionnement :

- Trouver un arbre couvrant de poids minimum T de G.
- Considérer V_1 l'ensemble des sommets de degré impair de T.
- Trouver un couplage parfait de poids minimum M de $G[V_1]$.
- Considérer le multigraphe H obtenu par union des arêtes de T et de M.
- Trouver un circuit eulérien C dans H.
- Transformer C en cycle hamiltonien en « sautant » les sommets répétés.

III.1 Recherche d'un arbre couvrant de poids minimum

1. Quel algorithme peut-on utiliser pour trouver un arbre couvrant de poids minimum dans un graphe ? L'appliquer sur G_{ex} en donnant les arêtes d'un arbre couvrant de poids minimum T_{ex} obtenu par l'algorithme, dans l'ordre.

III.2 Sommets de degré impair

Soit V_1 l'ensemble des sommets de degré impair dans T (en ne considérant que les arêtes de T).

2. Donner V_1 dans le cas où $T = T_{ex}$.
3. Montrer qu'un graphe G' non orienté possède un nombre pair de sommets de degré impair.

La question précédente montre donc que V_1 est de cardinal pair.

III.3 Couplage parfait de poids minimum

On définit $G[V_1]$ (**graphe induit** par V_1) comme le graphe dont l'ensemble de sommets est V_1 et dont deux sommets sont adjacents si et seulement si ils sont adjacents dans G. Les poids des arêtes de $G[V_1]$ sont les mêmes que ceux de G.

Un **couplage parfait de poids minimum** est un couplage parfait dont le poids est minimum parmi tous les couplages parfaits possibles.

4. Montrer l'existance d'un couplage parfait de poids minimum de $G[V_1]$.

5. On suppose que les sommets de G sont des points de $\mathbb{R}^2$ (comme c'est le cas pour G_{ex}). On peut alors considérer chaque arête comme un segment dans $\mathbb{R}^2$. Montrer que si M est un couplage de G dont deux arêtes se croisent (c'est-à-dire que les segments correspondants s'intersectent), alors M n'est pas de poids minimum.

6. Avec $G = G_{ex}$ et $T = T_{ex}$, donner les arêtes d'un couplage parfait de poids minimum M_{ex} de $G[V_1]$, ainsi que son poids.

III.4 Cycle eulérien

Un **multigraphe** est comme un graphe, sauf qu'il peut y avoir plusieurs arêtes entre deux sommets.
On définit le multigraphe $H = (V, E_H)$ dont les sommets sont les mêmes que G et tel que E_H contienne l'union des arêtes de T et de M (avec répétition : E_H est un multiensemble).

7. Dessiner H dans le cas où $G = G_{ex}$, $T = T_{ex}$ et $M = M_{ex}$.

Dans un graphe G' non orienté, un **cycle eulérien** est un cycle passant par chaque arête exactement une fois (mais qui peut passer plusieurs fois par un sommet, contrairement à un cycle hamiltonien).

8. Donner un cycle eulérien dans H dans le cas où $G = G_{ex}$, $T = T_{ex}$ et $M = M_{ex}$.

9. Montrer que si G' possède un cycle eulérien C, alors G' est connexe et tous les sommets de G' sont de degré pair.

On veut écrire un algorithme en OCaml permettant de trouver un cycle eulérien dans un graphe.

10. Écrire une fonction `supprime e l` supprimant la première occurence de `e` dans la liste `l`. Si `e` n'apparaît pas dans `l`, on renverra `l` inchangée. Par exemple, `supprime 3 [6; 3; 2; 3; 5]` doit renvoyer `[6; 2; 3; 5]`.

Dans la suite, on suppose que G' est connexe et que tous les sommets de G' sont de degré pair.

11. On part d'un sommet u quelconque de G', puis, tant que possible, on se déplace suivant une arête adjacente à u qui n'a pas encore été utilisée. Montrer que ce processus termine et que la liste des sommets visités forme un cycle C.

12. Écrire une fonction `cycle g u` qui renvoie le cycle C obtenu par l'algorithme précédent sous forme d'une liste de sommets, où G' est représenté par une liste d'adjacence `g`. On supprimera de `g` les arêtes utilisées par le cycle.

13. Montrer que G' possède un cycle eulérien, en raisonnant par récurrence.

14. En déduire une fonction `euler g u` qui renvoie un cycle eulérien (sous forme d'une liste de sommets) en partant depuis le sommet `u` dans le graphe G' donné sous forme de liste d'adjacence.

III.5 Cycle hamiltonien

Le cycle eulérien C dans le graphe H de la section précédente peut passer plusieurs fois par le même sommet. Pour le transformer en cycle hamiltonien, on supprime les sommets apparaissant plusieurs fois dans C (à part la première occurence).

15. Donner le cycle hamiltonien obtenu avec $G = G_{ex}$ ainsi que son poids. Comparer avec le cycle hamiltonien de poids minimum.

16. Écrire une fonction `supprime_doublons l` qui renvoie la liste obtenue en supprimant les doublons dans la liste de sommets `l`, sauf la première occurence de chaque sommet.
Par exemple, `supprime_doublons [1; 0; 1; 4; 6; 4]` doit renvoyer `[1; 0; 4; 6]`.

III.6 Preuve de la $\frac{3}{2}$-approximation

Soit C^* un cycle hamiltonien de poids minimum de G et T un arbre couvrant de poids minimum de G.

17. Montrer que $w(T) \leq w(C^*)$.

Soit V_1 l'ensemble des sommets de degré impair de T. Soit C_1^* un cycle hamiltonien de poids minimum de $G[V_1]$.

18. Montrer qu'on peut séparer les arêtes de C_1^* en deux couplages parfaits M_1 et M_2 de $G[V_1]$.

19. Soit M un couplage parfait de poids minimum de $G[V_1]$. Montrer que $w(M) \leq \min(w(M_1), w(M_2))$.

20. Montrer que $\min(w(M_1), w(M_2)) \leq \dfrac{w(C_1^*)}{2}$.

21. Montrer que $w(C_1^*) \leq w(C^*)$.

22. Soit C le cycle hamiltonien obtenu par l'algorithme de Christofides. Montrer que $w(C) \leq \dfrac{3}{2} w(C^*)$.