DS3-Durée 4h

04 décembre 2025

Ce sujet est composé de deux exercices et d'un problème.

1 Décidabilité

Pour chacun des problèmes A suivants, déterminer, en justifiant, s'il est décidable.

1. ARRÊT :
   - ⋆ **Instance** : un programme p et un argument x.
   - ⋆ **Question** : l'exécution de p sur x termine-t-elle ?
2. $\text{ARRÊT}_{\geq 10}$:
   - ⋆ **Instance** : un programme p.
   - ⋆ **Question** : le nombre d'arguments x pour lesquels l'exécution de p sur x se termine est-il supérieur ou égal à 10 ?
3. SOMME :
   - ⋆ **Instance** : une fonction OCAML $f : int \to int \to int \to bool$.
   - ⋆ **Question** : $f\ a\ b\ c$ renvoie-t-elle $true$ lorsque $a + b = c$?

2 Jeux dans un graphe

Un **graphe** (fini et orienté) est un couple $G = (S, A)$ tel que S est un ensemble fini non vide et $A \subseteq S \times S$. Pour $s \in S$ un sommet, on note $V(s)$ l'ensemble des **voisins** de s, c'est-à-dire $V(s) = \{t \in S \mid (s,t) \in A\}$. On appelle **chemin** σ une suite non vide, finie ou infinie, de sommets $(s_i)_{0 \leq i < m}$, avec $m \in \mathbb{N}^* \cup \{\infty\}$, telle que $s_{i+1} \in V(s_i)$ si $0 \leq i$ et $i + 1 < m$. Si $m \neq \infty$, on dit que σ est **fini de longueur** $m - 1$. Sinon, il est dit **infini**. Il est à noter qu'un chemin σ peut passer plusieurs fois par le même sommet. On notera $\mathcal{C}(G)$ l'ensemble des chemins (finis ou infinis) de G.

Pour $s \in S$, le **nombre d'occurrences de** s **dans** σ, noté $|\sigma|_s$, correspond aux nombres de fois qu'un sommet de T apparaît dans σ. Formellement, $|\sigma|_s = \text{Card}\,\{i \in [\![0, m[\![\mid s_i = s\}$, ce cardinal pouvant être fini ou infini. On note de même, pour $T \subseteq S$, $|\sigma|_T = \text{Card}\,\{i \in [\![0, m[\![\mid s_i \in T\}$.

On considère un jeu à deux joueurs dans un graphe. Informellement, une partie se déroule comme suit : un jeton est placé sur un sommet du graphe G et déplacé par les joueurs de sommets en sommets, par une succession de coups. Un coup consiste à déplacer le jeton en suivant une arête : lorsque le jeton est sur un sommet s, le joueur à qui appartient s le déplace sur un voisin de s, et ainsi de suite. Une partie est un chemin traversé par le jeton.

Formellement, une **arène** est un triplet (G, S_1, S_2) tel que $G = (S, A)$ est un graphe et $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$. Un **jeu** est un quadruplet (G, S_1, S_2, W) tel que (G, S_1, S_2) est une arène et $W \subset \mathcal{C}(G)$. Une **partie depuis un sommet initial** s est un chemin $\sigma = (s_i)_{0 \leq i < m}$ tel que $s_0 = s$. On dit que la partie σ est **gagnée** par le joueur 1 si $\sigma \in W$. Sinon, σ est dite gagnée par le joueur 2.

Pour $j \in \{1, 2\}$, une **stratégie pour le joueur** j est une application $f : S_j \to S$ telle que pour tout $s \in S_j$, $f(s) \in V(s)$. Une partie $\sigma = (s_i)_{0 \leq i < m}$ est dite une f**-partie** si pour tout $s_i \in S_j$, tel que $i + 1 < m$, $s_{i+1} = f(s_i)$. Une stratégie f pour le joueur j est dite **gagnante depuis** s si toute f-partie depuis s est gagnée par le joueur j. Un sommet $s \in S$ est dit gagnant pour j s'il existe une stratégie gagnante pour j depuis s.

Un jeu est dit **positionnel** si tout sommet est gagnant pour 1 ou 2, c'est-à-dire s'il existe $R_1, R_2 \subseteq S$ tels que $S = R_1 \cup R_2$ et deux stratégie f_1 et f_2 telles que f_j est gagnante pour j depuis tout sommet de R_j, pour $j \in \{1, 2\}$. On remarquera que R_1 (resp. R_2) peut contenir à la fois des sommets de S_1 et des sommets de S_2.

On considèrera dans l'ensemble de cette partie que les graphes considérés sont sans puits, c'est-à-dire que pour tout $s \in S$, $V(s) \neq \emptyset$.

2.1 Préliminaires

On considère l'arène de la figure 2.1, où $S_1 = \{1\}$ et $S_2 = \{0, 2\}$ (les sommets de S_1 sont représentés par des cercles, ceux de S_2 par des carrés).

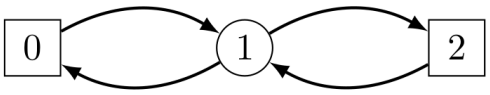


Figure 1 – Une arène (G, S_1, S_2).

Q 1. Donner toutes les stratégies possibles pour chacun des deux joueurs.
Q 2. On suppose que W est l'ensemble des chemins ne visitant pas le sommet 0 : $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_0 = 0\}$. Le jeu (G, S_1, S_2, W) est-il positionnel ? Si oui, donner les ensembles R_1 et R_2 et les stratégies f_1 et f_2 correspondants. Sinon, justifier.
Q 3. Même question si $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_0 = \infty \text{ et } |\sigma|_2 = \infty\}$.
Q 4. Montrer que si un jeu est positionnel, alors les ensembles R_1 et R_2 sont disjoints.

2.2 Jeux d'accessibilité

On suppose dans cette partie que $T \subseteq S$ et que $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_T > 0\}$. On représente un graphe $G = (S, A)$ en OCaml par un tableau de listes d'adjacence.

```
type graphe = int list array;;
```

Si `g` est une variable de type `graphe` correspondant à un graphe $G = (S, A)$, alors :

— $S = [\![0, n-1]\!]$, où $n =$ `Array.length g` ;

— pour $s \in S$, `g.(s)` est une liste contenant les éléments de $V(s)$ (sans doublons, dans un ordre arbitraire).

Q 5. Écrire une fonction `transpose : graphe -> graphe` qui prend en argument un graphe $G = (S, A)$ et renvoie son graphe transposé $G^T = (S, A')$ où $A' = \{(s, t) \mid (t, s) \in A\}$. On garantira une complexité en $(|S| + |A|)$ mais on ne demande pas de le justifier.

Une partie $T \subseteq S$ sera représentée par un tableau de booléens `tab` de taille n tel que `tab.(s)` vaut `true` si et seulement si $s \in S$.

Q 6. Dans cette question, on suppose que le jeu est à un seul joueur, c'est-à-dire que $S_2 = \emptyset$. Donner une condition nécessaire et suffisante pour qu'il existe une stratégie gagnante depuis s dans ce cas particulier (inutile de justifier). Écrire une fonction `strategie : graphe -> bool array -> int array` qui prend en argument un graphe G et une partie $T \subseteq S$ et renvoie un tableau `f` de taille n tel que pour tout $s \in S$:

— si $s \in T$, alors `f.(s)` $= n$;

— sinon, s'il existe une stratégie gagnante f_1 depuis s, alors `f.(s)` $= f_1(s)$;

— sinon, `f.(s)` $= -1$.

On garantira une complexité en $(|S| + |A|)$ et on demande de justifier cette complexité.

On suppose pour la suite que $S_1 \neq \emptyset$ et $S_2 \neq \emptyset$. Pour $X \subseteq S$, on définit par induction l'ensemble $\text{Attr}_i(X)$, pour $i \in \mathbb{N}$, par :

— $\text{Attr}_0(X) = X$;

— $\text{Attr}_{i+1}(X) = \text{Attr}_i(X) \cup \{s \in S_1 \mid V(s) \cap \text{Attr}_i(X) \neq \emptyset\} \cup \{s \in S_2 \mid V(s) \subseteq \text{Attr}_i(X)\}$.

Q 7. Montrer qu'il existe $k \in \mathbb{N}$ tel que $\bigcup_{i \in \mathbb{N}} \text{Attr}_i(X) = \text{Attr}_k(X)$. On notera $\text{Attr}(X)$ cet ensemble pour la suite et on l'appellera **attracteur de** X.

Q 8. Le jeu est positionnel. Plus précisément (compléter sans justifier) :

1. Le joueur 1 a une stratégie gagnante depuis $R_1 = \dots$.
2. Le joueur 2 a une stratégie gagnante depuis $R_2 = \dots$.

Expliquer comment contruire les stratégies gagnantes dans chacun des deux cas (on n'attend pas de preuve du fait qu'elles soient effectivement des stratégies gagnantes).

On rappelle que les files peuvent être utilisées en OCaml avec les commandes suivantes, toutes en complexité (1) :

— `Queue.create : unit -> 'a Queue.t` permet de créer une file vide ;

— `Queue.is_empty : 'a Queue.t -> bool` permet de tester si une file est vide ;

— `Queue.push : 'a -> 'a Queue.t -> unit` ajoute un élément à la fin d'une file ;

— `Queue.pop : 'a Queue.t -> 'a` enlève un élément au début d'une file et le renvoie.

Q 9. Écrire une fonction `attracteur : graphe -> bool array -> bool array -> bool array` qui prend en argument un graphe G, un tableau représentant $S_1 \subset S$ et un tableau représentant une partie $T \subseteq S$ et renvoie un tableau `attr` de taille n représentant Attr(T). Donner la complexité de la fonction.

2.3 Jeux de Büchi

On suppose dans cette partie que $T \subseteq S$ et que $W = \{\sigma \in \mathcal{C}(G) \mid |\sigma|_T = \infty\}$, c'est-à-dire qu'une partie est gagnée par le joueur 1 si elle visite infiniment souvent un sommet de T.

Q 10. Dans cette question, on suppose que le jeu est à un seul joueur, c'est-à-dire que $S_2 = \emptyset$. On considère $s \in S$. Donner une condition nécessaire et suffisante en termes d'existence de chemin et de cycle pour qu'il existe une stratégie gagnante depuis s. Avec quelle complexité temporelle peut-on calculer tous les sommets s tels qu'il existe une stratégie gagnante depuis s ? Détailler.

3 Le voyageur de commerce

On considère des graphes non orientés $G(V, E)$ où V est l'ensemble des sommets et E l'ensemble des arêtes. On notera $n = |V|$ le nombre de sommets.

Un *chemin* est une suite de sommets reliés par des arêtes. On dit qu'un chemin *passe* par un sommet si ce sommet appartient au chemin. Un *circuit* est un chemin qui commence et se termine au même sommet. Un *chemin hamiltonien* est un chemin qui passe une et une seule fois par chaque sommet du graphe. Un *circuit hamiltonien* est un circuit qui passe par chaque sommet une et une seule fois.

Le *problème du voyageur de commerce* consiste, étant donnée une liste de villes toutes reliées entre elles, à trouver le circuit le plus court qui passe une et une seule fois par chacune des villes.

Plus formellement, on considère un graphe complet non orienté, dont les arêtes sont étiquetées avec des nombres entiers strictement positifs, appelés *poids*, et on cherche le circuit passant par chacun des sommets du graphe qui minimise la somme des poids des arêtes. On appellera *poids d'un circuit* la somme des poids des arêtes empruntées par ce circuit. Une solution au problème du voyageur de commerce est un circuit hamiltonien de poids minimal.

Dans cette partie, on représente les graphes en `C` par des matrices d'adjacence. Le poids d'une arête est représenté par un `int`, une arête absente étant représentée par un 0. On représente un graphe par une structure de données avec deux attributs : son nombre de sommets `V` et un pointeur `adj` vers sa matrice d'adjacence de taille `V`×`V`. Les sommets sont associés aux entiers de 0 à `V`-1.

```
struct Graphe {
    int V;
    int* adj;
};
```

Pour accéder au poids de l'arête entre le sommet `i` et le sommet `j` du graphe `G`, on pourra utiliser l'expression `G.adj[i * G.V + j]`.

On pourra par la suite utiliser les deux fonctions suivantes, qui sont supposées travailler en temps constant :

```
struct Graphe alloue_graphe(int V) {
    int* adj = malloc(V * V * sizeof(int));
    struct Graphe g = {.V = V, .adj = adj};
return g;
}

void libere_graphe(struct Graphe g) {
    free(g.adj);
}
```

La fonction `alloue_graphe` prend comme argument un nombre `V` et renvoie un graphe qui possède `V` sommets et dont la matrice d'adjacence est allouée sur le tas, grâce à la fonction `malloc`. La fonction `libere_graphe` libère la mémoire du tas occupée par la matrice d'adjacence d'un graphe dont on n'a plus besoin.

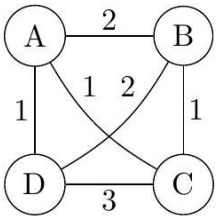
On définit également une structure `Chemin` qui représente un chemin par un attribut `longueur` et un attribut `l_sommets`, pointeur vers un tableau à `longueur` éléments. Si `C` est un chemin et `k` un entier naturel strictement plus petit que `C.longueur`, alors `C.l_sommets[k]` est le `k`-ième sommet du chemin `C`. De même que pour les graphes, on définit aussi des fonctions permettant d'allouer un chemin et de libérer un chemin dont on n'a plus besoin, et qui sont supposées travailler en temps constant.

```
struct Chemin {
    int longueur;
    int* l_sommets;
};

struct Chemin alloue_chemin(int longueur) {
    int* l_sommets = malloc(longueur * sizeof(int));
    struct Chemin c = {.longueur = longueur, .l_sommets = l_sommets};
return c;
}

void libere_chemin(struct Chemin c) {
    free(c.l_sommets);
}
```

1.Prise en main du problème et appartenance à NP.



Exemple d'instance du problème du voyageur de commerce

Q 11. Donner une solution du problème du voyageur de commerce sur l'exemple de la figure 1 en précisant le poids du circuit trouvé.
Q 12. Donner le nombre de circuits hamiltoniens sur un graphe complet de n sommets. Donner un exemple de pondération pour que chacun de ces circuits ait un poids minimal pour le problème du voyageur de commerce.
Q 13. Écrire une fonction

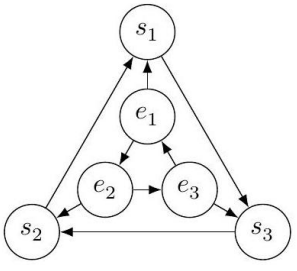
```
int poids_chemin(struct Graphe g, struct Chemin c);
```

qui prend en arguments un graphe et un chemin et renvoie le poids de ce chemin. Donner la complexité de cette fonction.
Q 14. Le problème du voyageur de commerce est un problème d'optimisation ; à l'aide d'un seuil, transformer ce problème en un problème de décision. Montrer que ce nouveau problème, que nous appellerons par la suite « problème de décision du voyageur de commerce » , appartient à la classe de complexité NP.

2.Étude de la complexité

Le *problème du chemin hamiltonien* consiste, étant donné un graphe non orienté et deux sommets a et b, à déterminer s'il existe un chemin hamiltonien commençant en a et finissant en b. Le *problème du chemin hamiltonien orienté* consiste, étant donné un graphe orienté et deux sommets a et b, à déterminer s'il existe un chemin hamiltonien commençant en a et finissant en b. Le *problème du circuit hamiltonien* consiste, étant donné un graphe non orienté, à déterminer s'il existe un circuit hamiltonien dans ce graphe.
Q 15. Montrer que le problème du chemin hamiltonien se réduit au problème du circuit hamiltonien.
Q 16. Montrer que le problème du circuit hamiltonien se réduit au problème de décision du voyageur de commerce.
Q 17. Montrer que le problème du chemin hamiltonien orienté se réduit au problème du chemin hamiltonien.

Le problème 3-SAT consiste à déterminer la satisfiabilité d'une formule logique sous forme normale conjonctive avec exactement 3 littéraux : pour n clauses C_i et m variables y_k, déterminer s'il existe une valuation des y_k qui permette de rendre vraie la formule $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ avec $C_i = x_{i,1} \vee x_{i,2} \vee x_{i,3}$ sachant que, pour chaque i et chaque p, il existe un k tel que $x_{i,p} = y_k$ ou $x_{i,p} = \neg y_k$. On admet que le problème 3-SAT est NP-complet. On va maintenant montrer que 3-SAT se réduit au problème du chemin hamiltonien orienté.



Graphe A

Q 18. On considère le graphe A (figure 2). Montrer qu'il existe un chemin entrant par e_1 et passant par tous les sommets pour ressortir en s_1. Puis qu'il existe un chemin entrant par e_1 et sortant par s_1 et un chemin entrant par e_2 et sortant par s_2, tels que chaque sommet soit visité par un et un seul des deux chemins. Même question, mais avec trois chemins, pour e_1, e_2, e_3 et s_1, s_2, s_3.

On se donne une instance du problème 3-SAT, pour n clauses C_i et m variables y_k : $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ avec $C_i = x_{i,1} \vee x_{i,2} \vee x_{i,3}$ et $x_{i,p} = y_k$ ou $x_{i,p} = \neg y_k$. On veut donc savoir s'il existe une valuation des y_k pour que la formule soit vraie.

On construit alors le graphe orienté G de la manière suivante :

— pour chaque variable y_k on crée un sommet v_k ;

— on ajoute un sommet supplémentaire v_{m+1} ;

— pour chaque clause C_i on ajoute une copie du graphe A, notée A_i ;

— pour chaque variable y_k, on note $C_{k_1}, \ldots, C_{k_\ell}$ les clauses dans lesquelles y_k apparait en positif. On relie alors v_k à A_{k_1} par un arc allant de v_k vers e_p dans A_{k_1} lorsque y_k est en position p dans C_{k_1} c'est-à-dire lorsque $C_{k_1} = x_{k_1,1} \vee x_{k_1,2} \vee x_{k_1,3}$ et $x_{k_1,p} = y_k$. On relie ensuite la sortie s_p de A_{k_1} à l'entrée de A_{k_2} correspondant à la position de y_k dans C_{k_2} et ainsi de suite jusqu'au dernier dont on relie la sortie à v_{k+1}. On appelle G_k^+ le sous-graphe constitué du sommet v_k, des graphes $A_{k_1}, A_{k_2}, \ldots, A_{k_\ell}$ et du sommet v_{k+1}, ainsi que des arcs que l'on vient d'ajouter entre eux en considérant y_k et les clauses dans lesquelles il apparait positivement ;

— on crée de même des arcs pour chaque variable y_k et chaque clause dans laquelle y_k apparait en négatif. On note G_k^-, le sous-graphe correspondant entre v_k et v_{k+1}.

Q 19. Montrer que pour toute valuation de la formule, qui la rend vraie, il existe un chemin hamiltonien orienté de v_1 à v_{m+1} dans le graphe G.

Q 20. Montrer, en une dizaine de lignes au maximum, que pour chaque chemin hamiltonien orienté de v_1 à v_{m+1} il existe bien une valuation.

Q 21. En déduire que le problème du circuit hamiltonien et le problème de décision du voyageur de commerce sont NP-complets (on pourra répondre soigneusement à cette question en admettant les précédentes même si on ne les a pas réussies).

3.Difficulté de l'approximation.

Soit $\varepsilon > 0$, on va montrer que, si P$\neq$NP, il n'existe pas de $1+\varepsilon$ approximation pour le problème du voyageur de commerce. Un algorithme est une $1+\varepsilon$ approximation à un problème d'optimisation d'un poids p lorsque la solution proposée de poids p se compare toujours à la solution optimale p^* par $p < (1+\varepsilon)p^*$.

Soit G un graphe non orienté à n sommets, on considère le graphe complet G', de mêmes sommets que G, avec des poids aux arêtes obtenus en donnant un poids de 1 aux arêtes de G et un poids de $n(1+\varepsilon)+1$ aux arêtes qui ne sont pas dans G.

Q 22. Montrer que si G possède un circuit hamiltonien, alors G' possède un circuit de poids n.

Q 23. Montrer que si G ne possède pas de circuit hamiltonien alors toute solution pour l'instance de voyageur de commerce est de poids au moins $n(2+\varepsilon)$.

Q 24. En déduire que, si P$\neq$NP, il n'existe pas de $1+\varepsilon$ approximation au problème du voyageur de commerce.

4. Algorithme de Christofides.

On va proposer une heuristique pour le problème du voyageur de commerce, l'algorithme de Christofides, et on va montrer que, sous certaines conditions sur le graphe en entrée, cette heuristique constitue un algorithme d'approximation. L'algorithme prend en argument un graphe G et procède comme suit :

— calculer un arbre couvrant de poids minimal T de G ;

— en notant I l'ensemble des sommets de degré impair dans T, calculer un couplage parfait M de poids minimum dans le sous-graphe de G induit par les sommets de $I, G_{|I}$;

— construire H le multigraphe ayant pour sommets les sommets de G et comme arêtes les arêtes de M et celles de T ;

— trouver un cycle eulérien dans H ;

— transformer le cycle eulérien en circuit hamiltonien en supprimant les éventuels sommets vus plusieurs fois.

Dans la suite, on étudie plus précisément certaines étapes de cet algorithme, avant de proposer une implémentation de cet algorithme.

Un arbre couvrant est un sous-graphe connexe sans cycle d'un graphe avec les mêmes sommets. On appelle poids de l'arbre la somme des poids des arêtes de cet arbre. On rappelle que l'algorithme de Kruskal est un algorithme glouton qui vise à construire un arbre couvrant de poids minimal en considérant les arêtes par poids croissant et en ajoutant chaque arête si elle ne crée pas de cycle.

Pour cela, on va représenter une arête par une structure de données avec trois attributs : `s1` et `s2` donnent les sommets reliés par l'arête et `p` son poids :

```
struct Arete {
int s1;
int s2;
int p;
};
```

Q 25. Écrire une fonction
`struct Arete* liste_aretes (struct Graphe g);`
qui prend en argument un graphe complet `g` et alloue et renvoie un tableau contenant les $\frac{g.V \times (g.V-1)}{2}$ arêtes du graphe.

On dispose d'une fonction `void tri_aretes(struct Arete a[], int k);` qui prend en arguments un tableau d'arêtes et sa longueur et trie le tableau par ordre croissant de poids. La complexité de cette fonction est en $\mathcal{O}(k \ln(k))$.

Q 26. Écrire une fonction `struct Graphe kruskal(struct Graphe g);` implémentant l'algorithme de Kruskal qui renvoie un graphe représentant l'arbre couvrant de poids minimal du graphe donné en argument. On mettra des 0 lorsque l'arête est absente et des 1 lorsqu'elle est présente. Donner la complexité de la fonction `kruskal`.

Q 27. Montrer la correction de cet algorithme, c'est-à-dire l'optimalité de la solution proposée pour le problème d'arbre couvrant de poids minimal.

On appelle couplage d'un graphe un ensemble d'arêtes qui n'ont pas de sommets en commun. Un couplage est parfait si tous les sommets du graphe appartiennent à une arête du couplage.

Q 28. Écrire une fonction `int degré(struct Graphe g, int i);` qui prend en arguments un graphe et l'indice d'un sommet et renvoie le degré de ce sommet.

Q 29. Écrire une fonction `int* sommets_impairs(struct Graphe g, int* nb_sommets);` qui prend en arguments un graphe `g` et un pointeur vers un entier. Cette fonction alloue sur le tas un tableau, le remplit avec les numéros des sommets de degré impair et le renvoie. Par ailleurs, elle renseigne l'entier pointé par `nb_sommets` avec le nombre des sommets de degré impair.

Q 30. Montrer l'existence d'un couplage parfait de poids minimal dans $G_{|I}$.

On dispose des deux fonctions suivantes :
`struct Graphe graphe_induit(struct Graphe g, int nb_sommets, int* liste_sommets);`
`struct Graphe couplage(struct Graphe g);`
La fonction `graphe_induit` renvoie le graphe induit dans un graphe `g` donné par un nombre et un tableau de sommets comme ceux de la question 19.

La fonction `couplage` renvoie un couplage parfait de poids minimal s'il existe, sous la forme d'un graphe représentant ce couplage, avec des 0 lorsque l'arête est absente et 1 lorsque l'arête est présente.

On supposera ces deux fonctions de complexité polynomiale.

On appelle cycle eulérien un circuit qui passe par chaque arête une et une unique fois. On admet qu'un graphe possède un cycle eulérien si et seulement les degrés des sommets de ce graphe sont pairs et que le graphe est connexe. Un multigraphe est un graphe dans lequel il peut exister plusieurs arêtes reliant un même couple de sommets.

Q 31. Montrer que le multigraphe H, défini dans l'introduction de la sous-partie I.C, possède un cycle eulérien.

On dispose des trois fonctions suivantes :

```
struct Multigraphe multigraphe(struct Graphe g1, struct Graphe g2);
struct Chemin eulerien(struct Multigraphe h);
void libere_multigraphe(struct Multigraphe h);
```

La fonction `multigraphe` prend en arguments deux graphes sur les mêmes sommets et renvoie le multigraphe obtenu en considérant les arêtes des deux graphes. La fonction `eulerien` renvoie un circuit eulérien d'un multigraphe sous la forme d'un chemin. La fonction `libere_multigraphe` libère la mémoire du tas utilisée par un multigraphe. Toutes trois sont supposées de complexité polynomiale.

Q 32. Écrire une fonction `struct Chemin euler_to_hamilton(struct Chemin c);` qui transforme un cycle eulérien `c` du multigraphe H en un circuit hamiltonien du graphe G en supprimant les doublons, et renvoie le chemin représentant la suite des sommets.
Q 33. Sur l'exemple de la figure 1, réaliser les différentes étapes de l'algorithme. On ne demande pas de détailler les étapes pour trouver un couplage et un cycle eulérien.
Q 34. Écrire une fonction `struct Chemin christofides(struct Graphe g);` qui implémente l'algorithme de Christofides, en prenant soin de libérer la mémoire allouée sur le tas qui n'est plus utilisée.
Q 35. Justifier que la fonction `christophides` renvoie bien un circuit hamiltonien.
Q 36. Montrer que la fonction `christophides` est de complexité polynomiale.