CONCOURS CENTRALE•SUPÉLEC

Informatique

MPI 2023

4 heures Calculatrice autorisée

Le sujet est composé de deux parties indépendantes, la première utilise le langage C et la seconde le langage OCaml. Toutes deux traitent de graphes.

La première se concentre sur un problème classique, le problème du voyageur de commerce. Elle étudie d'abord sa complexité, en le comparant notamment au problème très proche du cycle et du chemin hamiltonien, puis en le réduisant à 3-SAT, et en donnant un résultat sur l'approximabilité du problème. On propose ensuite une heuristique, l'algorithme de Christofides, qui repose sur différentes notions : arbres couvrants, couplages, circuit eulérien. On montrera que, dans une variante du problème où les distances vérifient l'inégalité triangulaire, l'algorithme de Christofides est une approximation de facteur 3/2. Dans cette première partie, les graphes seront représentés par des matrices d'adjacence en C.

La deuxième partie s'intéresse à des processus d'édition sur des arbres non racinés dont les feuilles sont étiquetées, qui sont notamment utilisés en bio-informatique pour représenter l'évolution des espèces. On étudiera en particulier l'espace induit sur ces arbres par ces processus d'édition et on représentera cet espace par un graphe. On montrera notamment que le graphe induit par l'un de ces processus d'édition possède un cycle hamiltonien. On utilisera pour cette partie le langage OCaml pour implémenter des arbres par une définition récursive et des graphes par liste d'adjacence.

I Problème du 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);
}
```

I.A – Prise en main du problème et appartenance à NP

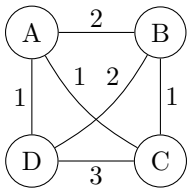


Figure 1 Exemple d'instance du problème du voyageur de commerce

Q 1. Donner une solution du problème du voyageur de commerce sur l'exemple de la figure 1 en précisant le poids du circuit trouvé.

Q 2. 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 3. 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 4. 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.

I.B – 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.

I.B.1) NP-complétude

Q 5. Montrer que le problème du chemin hamiltonien se réduit au problème du circuit hamiltonien.

Q 6. Montrer que le problème du circuit hamiltonien se réduit au problème de décision du voyageur de commerce.

Q 7. 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.

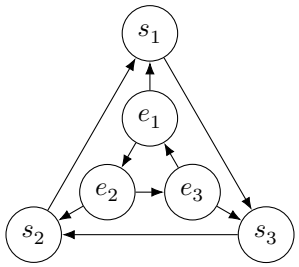


Figure 2 Graphe A

Q 8. 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 $C_{k_1}, ..., C_{k_\ell}$ 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}, ..., A_{k_ℓ} 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 9. Montrer que pour toute valuation de la formule il existe un chemin hamiltonien orienté de v_1 à v_{m+1} dans le graphe G.

Q 10. 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 11. En déduire que le problème du circuit hamiltonien et le problème de décision du voyageur de commerce sont NP-complets.

I.B.2) 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 12. Montrer que si G possède un circuit hamiltonien, alors G' possède un circuit de poids n.

Q 13. 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 14. En déduire que, si P $\neq$ NP, il n'existe pas de $1+\varepsilon$ approximation au problème du voyageur de commerce.

I.C – 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 sommet 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.

I.C.1) Arbre couvrant

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 15. 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 16. 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 17. 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.

I.C.2) Couplage

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 18. Écrire une fonction

```
int degre(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 19. 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 20. 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.

I.C.3) Cycle eulérien

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 21. 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 22. 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.

I.C.4) Implémentation

Q 23. 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 24. 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 25. Justifier que la fonction `christophides` renvoie bien un circuit hamiltonien.

Q 26. Montrer que la fonction `christophides` est de complexité polynomiale.

I.C.5) Preuve de l'approximation

On va maintenant montrer que cet algorithme est une 3/2-approximation pour le problème du voyageur de commerce, dans le cas où les poids des arêtes vérifient l'inégalité triangulaire, c'est-à-dire que pour tout sommet u, v, w les poids des arêtes vérifient, en notant $c_{i,j}$ le poids de l'arête entre des sommets i et j : $c_{u,v} \leqslant c_{u,w} + c_{w,v}$.

On note U une solution optimale et $c(U)$ son poids.

Q 27. Montrer que $c(T) \leqslant c(U)$, où $c(T)$ est le coût de l'arbre couvrant minimal.

Q 28. Montrer que $c(M) \leqslant 0.5c(U)$ où $c(M)$ est le coût du couplage parfait minimal.

Q 29. Montrer que la solution construite par l'algorithme est une 3/2-approximation de U.

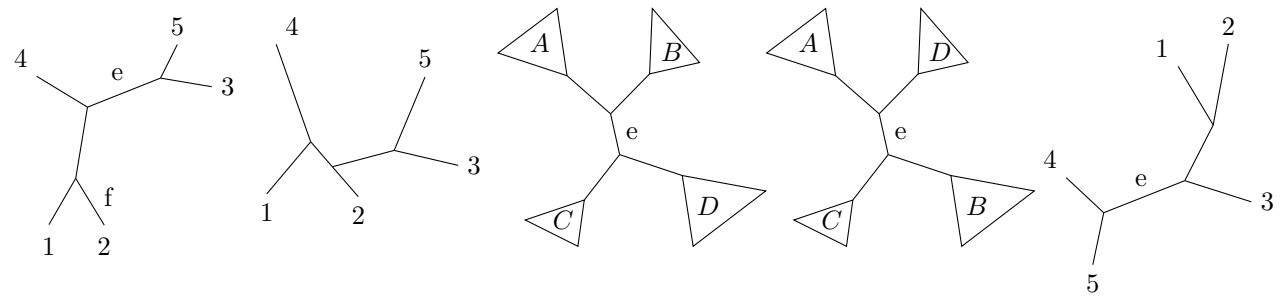
Q 30. Sans la supposition de l'inégalité triangulaire, cette solution est-elle toujours une 3/2-approximation ? Proposer un contre-exemple ou une justification.

II Espaces d'arbres

Dans cette partie on considère des arbres binaires non racinés avec des feuilles étiquetées. Un *arbre binaire non raciné* est un graphe connexe sans cycle dont les nœuds sont de degrés 1 ou 3. Un *arbre binaire raciné* est un graphe connexe sans cycle dont les nœuds sont de degrés 1 ou 3 sauf exactement un nœud qui est de degré 2. Les nœuds de degré 1 sont appelés des *feuilles*, les nœuds de degré 2 ou 3 des *nœuds internes* et l'unique nœud de degré 2 d'un arbre binaire raciné est appelé sa *racine*. Un graphe réduit à un unique nœud est considéré comme un arbre raciné dont le nœud est à la fois feuille et racine. Les arêtes d'un arbre sont appelées des *branches*, les branches reliant des sommets de degré 2 ou 3 *branches internes*. On étiquette ensuite les feuilles d'un arbre à n feuilles par des entiers distincts entre 1 et n. On s'intéresse à des opérations d'édition qui transforment un arbre à n feuilles en un second arbre à n feuilles.

La première opération, notée SPR, pour *Subtree Prune and Regraft* (ou découpe et greffe d'un sous-arbre), prend en entrée 2 branches e et f de l'arbre. La branche e est supprimée, créant deux arbres racinés par les sommets à chaque extrémité de e, notons ces arbres T_1 (qui contient la branche f) et T_2 (qui ne la contient pas). On divise ensuite la branche f pour créer un nouveau nœud v de degré 2. Le sous-arbre raciné T_2 est alors rattaché par sa racine à v par une nouvelle branche ; v devient alors de degré 3. On obtient alors un arbre raciné, avec une racine de degré 2, que l'on supprime en reliant directement ses deux voisins pour obtenir un arbre non raciné.

On s'intéresse également à une autre opération, notée NNI, pour *Nearest Neighbor Interchange* (ou échange entre plus proches voisins). Cette opération prend une branche interne de l'arbre e et échange entre eux deux des quatre sous-arbres incidents à cette branche. C'est-à-dire, en notant a et b les extrémités de la branche interne e, a a deux sommets voisins qui ne sont pas b, a_1 et a_2, et de même pour b, b_1 et b_2. On choisit alors un des a_i et un des b_j à échanger, par exemple a_1 et b_1, on supprime l'arête reliant a à a_1 et b à b_1 et on relie a à b_1 et b à a_1.



Arbre 1 Arbre 2 Arbre 3 Arbre 4 Arbre 5

Figure 3 Exemples d'arbres non racinés

La figure 3 présente quelques exemples d'arbres non racinés :

— l'arbre 1 est un arbre binaire non raciné avec ses feuilles étiquetées ;

— l'arbre 2 est obtenu à partir de l'arbre 1 par un SPR en coupant la branche e et en la greffant sur la branche f ;

— dans l'arbre 3, où les triangles représentent des sous-arbres racinés, la branche interne e sépare 4 sous-arbres racinés A, B, C et D ;

— l'arbre 4 est obtenu à partir de l'arbre 3 par un mouvement NNI sur la branche e en échangeant les sous-arbres B et D ;

— l'arbre 5 est obtenu à partir de l'arbre 1 par un NNI sur la branche e en échangeant le sous-arbre contenant la feuille 5 et le sous-arbre contenant les feuilles 1 et 2.

On étudie l'espace des arbres binaires non racinés étiquetés de n feuilles $B(n)$ suivant ces deux processus d'édition. On définit $G_{\text{NNI}}(n)$ le graphe dont les sommets sont les arbres de $B(n)$ et dans lequel une arête relie deux arbres si l'on peut passer de l'un à l'autre par un mouvement NNI. On définit de même $G_{\text{SPR}}(n)$ avec les mouvements SPR. On va notamment s'intéresser à la structure de $G_{\text{NNI}}(5)$, écrire un programme permettant de construire $G_{\text{NNI}}(n)$ et montrer que $G_{\text{SPR}}(n)$ est hamiltonien, c'est-à-dire qu'il possède un circuit hamiltonien.

II.A – Prise en main

Q 31. Montrer que l'on peut passer de l'arbre 1 à l'arbre 2 par une opération NNI. Dessiner tous les voisins de l'arbre 1 dans $G_{\text{NNI}}(5)$.

Q 32. Donner le nombre de branches et de nœuds d'un arbre de $B(n)$, pour $n \geqslant 2$.

Q 33. On note $RB(n)$ l'ensemble des arbres binaires racinés à n feuilles étiquetées. Montrer que $|RB(n)| = (2n-3)|B(n)|$ et que $|B(n)| = |RB(n-1)|$. En déduire le nombre d'arbres binaires $B(n)$ en fonction de n.

Q 34. Tracer $G_{\text{NNI}}(4)$.

Q 35. Combien de voisins un arbre de $B(n)$ a-t-il dans $G_{\text{NNI}}(n)$?

On appelle arbre chenille un arbre binaire non raciné dans lequel il existe un chemin passant une et une seule fois par chaque nœud interne.

Q 36. Montrer que l'on peut passer de tout arbre de $B(n)$ à un arbre chenille, en effectuant seulement des mouvements NNI. En déduire que $G_{\text{NNI}}(n)$ est connexe.

Q 37. Montrer que tout mouvement SPR peut se décomposer en mouvements NNI, et que tous les mouvements NNI sont des mouvements SPR. En déduire que $G_{\text{SPR}}(n)$ est connexe.

II.B – Étude de $G_{\text{NNI}}(5)$

Q 38. Combien de sommets possède $G_{\text{NNI}}(5)$ et quel est le degré de ces sommets ?

Q 39. Montrer que tout arbre de $G_{\text{NNI}}(5)$ fait partie de cycles de longueur 5, de deux cycles de longueurs 3, et que pour tout arbre deux arbres sont à distance 3 de cet arbre.

Q 40. Tracer $G_{\text{NNI}}(5)$.

II.C – Construction de $G_{\mathbf{NNI}}(n)$

On va écrire en OCaml un programme qui construit $G_{\mathrm{NNI}}(n)$. Pour représenter les arbres binaires (racinés ou non), nous allons utiliser la structure de données suivante :

```
type arbre = Feuille of int
           | Noeud of arbre * arbre ;;
```

Un arbre raciné est alors représenté par sa racine, qui peut être soit une feuille soit un nœud interne ayant un sous-arbre gauche et un sous-arbre droit.

Un arbre non raciné à n feuilles est représenté à partir d'une racine fictive ajoutée entre la feuille d'étiquette n et le nœud auquel elle est reliée.

Enfin, pour assurer l'unicité de la représentation informatique d'un arbre, et ainsi simplifier les programmes, on adopte la convention, pour chaque nœud interne, que la plus grande des étiquettes du sous-arbre droit est supérieure aux étiquettes du sous-arbre gauche.

Ainsi, l'arbre 1 de la figure 3 est représenté par :

```
Noeud (Noeud (Feuille 3,
              Noeud (Noeud (Feuille 1,
                            Feuille 2),
                     Feuille 4)),
       Feuille 5);;
```

On représente les graphes par liste d'adjacence, plus précisément par une liste de couples dont le premier élément est le nom d'un sommet et le second la liste d'adjacence de ce sommet.

```
type graphe = (arbre * arbre list) list;;
```

Q 41. Écrire une fonction `feuilles` qui prend en argument un arbre et renvoie la liste des étiquettes de ses feuilles.

Q 42. Écrire une fonction `degres` qui prend en argument un graphe implémenté par liste d'adjacence et renvoie la liste des degrés de ces nœuds.

Q 43. Écrire une fonction `egaux` qui teste si deux arbres sont égaux, au sens des arbres binaires non racinés étiquetés. Justifier que cette fonction est correcte.

Q 44. Étant donnés une liste d'arbres et un arbre, écrire une fonction `appartient` qui teste si l'arbre fait partie de la liste.

Q 45. En remarquant que parmi les 4 sous-arbres à échanger pour un mouvement de NNI autour d'une branche interne, on peut en choisir un qui restera fixe, écrire une fonction `voisinsNNI` qui prend en argument un arbre et renvoie tous les arbres que l'on peut obtenir à partir de celui-ci par un mouvement NNI.

Q 46. Écrire une fonction `chenille` qui prend en argument un entier n et renvoie l'arbre chenille dans lequel les feuilles sont rangées de 1 à n.

Q 47. Écrire une fonction `insere` qui prend en arguments un arbre et un graphe et ajoute l'arbre à la liste des sommets du graphe.

Q 48. Écrire une fonction `relie` qui prend en arguments un graphe et deux arbres et qui rajoute au graphe une arête reliant les deux sommets, si elle n'est pas déjà présente.

Q 49. Écrire une fonction `grapheNNI` qui prend en argument un entier n et renvoie $G_{\mathrm{NNI}}(n)$, en implémentant le graphe avec des listes d'adjacences. Justifier la correction et la terminaison de votre programme.

II.D – $G_{\mathbf{SPR}}(n)$ est hamiltonien

On va démontrer par récurrence que $G_{\mathrm{SPR}}(n)$ est hamiltonien, c'est-à-dire qu'il possède un circuit hamiltonien, comme défini en première partie.

On note S_i l'ensemble des arbres de taille $n+1$ obtenu à partir d'un arbre t_i de taille n en ajoutant une feuille étiquetée $n+1$ à une des branches de t_i. Pour ajouter une feuille à une branche, on supprime la branche et on crée un nouveau nœud, relié à la nouvelle feuille et aux deux nœuds qui étaient reliés par la branche.

Q 50. Montrer que les sous-graphes $G_{\mathrm{SPR}}(n+1)_{|S_i}$, induits dans $G_{\mathrm{SPR}}(n+1)$ par les sommets de S_i sont complets.

Q 51. Soient t_i et t_j deux sommets de $G_{\mathrm{SPR}}(n)$ reliés entre eux, montrer qu'il existe des arêtes entre S_i et S_j dans $G_{\mathrm{SPR}}(n+1)$.

Q 52. Démontrer par récurrence que $G_{\mathrm{SPR}}(n)$ est hamiltonien.

• • • FIN • • •