Partie I - Problème du voyageur de commerce

Cette partie comporte des questions nécessitant un code C.

Soit $G = (S, A)$ un graphe non orienté à $|S| = n$ sommets. On note ce graphe $G = (S, A, c)$ lorsqu'il est muni d'une fonction de valuation des arêtes $c : S \times S \to \mathbb{N}$ telle que pour tout $s \in S$, $c(s, s) = 0$ et $c(s, t) = +\infty$ s'il n'existe pas d'arête entre s et t.

Définition 1 (Cycle hamiltonien)
Un cycle hamiltonien dans $G = (S, A)$ est un chemin qui passe une fois et une seule par chaque sommet de G et qui commence et termine par le même sommet.

On considère les problèmes suivants :

- CYCLE-HAMILTONIEN qui, étant donné un graphe non orienté $G = (S, A)$, décide s'il existe un cycle hamiltonien dans G,
- COUVERTURE-SOMMET qui, étant donnés un graphe non orienté $G = (S, A)$ et $m \in \mathbb{N}$, décide s'il existe un sous-ensemble de sommets $S_c \subseteq S$ tel que $|S_c| \leq m$ et que chaque arête de A ait au moins une de ses extrémités dans S_c,
- TSP qui, étant donnés un graphe $G = (S, A, c)$ et un entier $k \in \mathbb{N}$, décide s'il existe un cycle hamiltonien dans G de coût inférieur ou égal à k.

On admet dans la suite que COUVERTURE-SOMMET est NP-complet. On souhaite alors montrer que TSP est NP-complet.

Q1. Montrer que TSP est dans NP.

I.1 - Réduction de CYCLE-HAMILTONIEN vers TSP

Q2. Montrer que CYCLE-HAMILTONIEN est dans NP.

Soit $G = (S, A)$ une instance de CYCLE-HAMILTONIEN. On construit l'instance correspondante de TSP comme suit :

(i). on crée un nouveau graphe $G' = (S', A')$ qui est un graphe complet tel que $S' = S$,

(ii). on attribue un coût à chaque arête de G' :
- si l'arête (s, t) existe dans G, on lui attribue un coût de 1,
- si l'arête (s, t) n'existe pas dans G, on lui attribue un coût de 2,

(iii). on fixe un seuil de coût $k = |S|$.

Q3. Montrer que la construction de G' est polynomiale en la taille de G.

Q4. Montrer que si CYCLE-HAMILTONIEN renvoie Vrai sur l'entrée G alors TSP renvoie Vrai sur l'entrée (G', k).

Q5. Montrer que si TSP renvoie Vrai sur l'entrée G' alors CYCLE-HAMILTONIEN renvoie Vrai sur l'entrée G.

I.2 - Réduction de COUVERTURE-SOMMET vers CYCLE-HAMILTONIEN

Soient une instance de COUVERTURE-SOMMET donnée par le graphe $G = (S, A)$ et un entier $m \leq |S|$. On doit construire un graphe $G' = (S', A')$ tel que G' a un cycle hamiltonien si et seulement si G a une couverture de sommets de taille au plus m. On admettra cette équivalence dans la suite. On s'intéresse, pour les questions **Q6** à **Q10**, uniquement à la construction de G' à partir de G. De plus, pour les questions **Q6** à **Q9**, on utilisera le graphe G décrit dans la **Figure 1**.

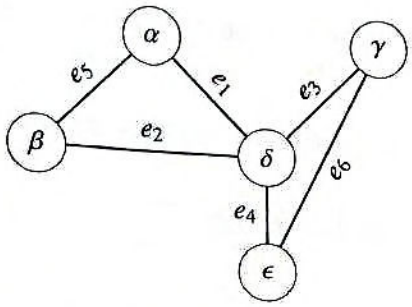


Figure 1 - Graphe exemple

Q6. Le problème COUVERTURE-SOMMET répond-il Vrai pour $m = 2$? pour $m = 3$? Si oui, donner pour chaque valeur de m un sous-ensemble S_c correspondant.

La construction de G' passe par les étapes suivantes :

(i). on construit m sommets $a_1 \dots a_m$ utilisés pour sélectionner m sommets de G,

(ii). pour chaque arête $e = (s,t)$ de G, on construit un graphe gadget $G_e = (S_e, A_e)$ à 12 sommets et 14 arêtes avec $S_e = \{s_i(e), i \in [\![1,6]\!]\} \cup \{t_i(e), i \in [\![1,6]\!]\}$ et

$$A_e = \left\{\{s_i(e), s_{i+1}(e)\}\right\}_{i\in[\![1,5]\!]} \cup \left\{\{t_i(e), t_{i+1}(e)\}\right\}_{i\in[\![1,5]\!]} \cup \left\{\{s_3(e), t_1(e)\}, \{t_3(e), s_1(e)\}\right\} \cup \left\{\{s_6(e), t_4(e)\}, \{t_6(e), s_4(e)\}\right\}.$$

(iii). pour chaque sommet $s \in S$ de degré $d(s)$, on connecte tous les graphes gadgets où s apparaît. Pour ce faire, on construit un ensemble d'arêtes $A_s = \{\{s_6(e_{s[j]}), s_1(e_{s[j+1]})\}, j \in [\![1, d(s)-1]\!]\}$, où les $e_{s[j]}$ sont les arêtes incidentes à s ordonnées arbitrairement. Cette étape créé un chemin dans G' composé exactement des sommets $u_i(e)$ tels que $u = s$.

(iv). on complète les arêtes de G' en connectant les sommets $a_1 \dots a_m$ aux premiers et derniers sommets de chaque chemin créé dans l'étape précédente. Pour ce faire, on construit les arêtes $A_c = \{\{a_i, s_1(e_{s[1]})\}, \{a_i, s_6(e_{s[d(s)]})\}, i \in [\![1,m]\!], s \in S\}$.

Le graphe G' est alors défini par $S' = \left\{a_i, i \in [\![1,m]\!]\right\} \cup (\cup_{e\in A} S_e)$ et $A' = (\cup_{e\in A} A_e) \cup (\cup_{s\in S} A_s) \cup A_c$.

Q7. Dessiner le graphe gadget de l'arête $e_2 = (\beta,\delta)$. Représenter les $\beta_i(e_2)$ sur une même ligne, les $\delta_i(e_2)$ sur une même ligne, chaque $\beta_i(e_2)$ étant à la verticale de $\delta_i(e_2)$.

Soit $e = (s,t)$ une arête de G. Le graphe gadget G_e permet de garantir qu'au moins une extrémité de e figure parmi les m sommets sélectionnés dans l'étape (i). Dans la construction finale de G', les seuls sommets de G_e qui seront impliqués dans des arêtes supplémentaires construites dans l'étape (ii) sont $s_1(e), t_1(e), s_6(e)$ et $t_6(e)$.

Q8. Donner alors les trois traversées possibles du gadget G_e par un cycle hamiltonien dans G'.

Q9. Donner les ensembles A_γ et A_δ produits par l'étape (iii).

Q10. Montrer que G' peut être construit à partir de G et m en temps polynomial.

I.3 - Conclusion des réductions

Q11. En déduire que TSP est NP complet.

I.4 - 2-approximation du problème TSP

On change ici la manière de voir le problème TSP. On passe du problème de décision au problème d'optimisation : étant donné un graphe $G = (S, A, c)$, trouver dans G le chemin de coût total minimal qui passe exactement une fois par chaque sommet et revient au sommet de départ.

On ajoute maintenant une contrainte sur la fonction c : on demande à ce qu'elle respecte l'inégalité triangulaire, c'est-à-dire : $\forall (s,t,u) \in S^3\ c(s,u) \leq c(s,t) + c(t,u)$. On suppose de plus que le graphe G est complet.
On montre alors, dans la suite, qu'il existe une 2-approximation du problème TSP. À cette fin, on propose l'**algorithme 1**.

Algorithme 1 - 2-approximation du problème TSP

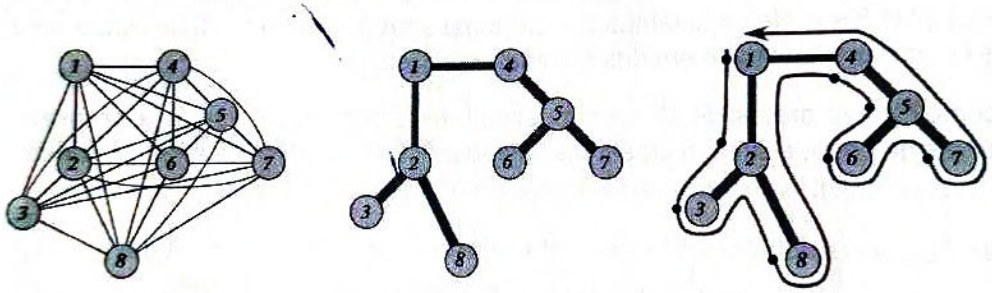
Entrées : $G = (S, A, c)$, $c : S \times S \to \mathbb{N}$
Sortie : Un cycle hamiltonien H
début
    T = arbre couvrant de coût minimal de G
    L = liste des sommets visités lors d'un parcours en profondeur de T
    H = cycle hamiltonien qui visite les sommets dans l'ordre de L.

Q12. Quel algorithme peut-on utiliser pour réaliser l'étape 2 de l'**algorithme 1** ? Quelle est la complexité de votre solution ?

On note H^* un cycle hamiltonien de coût optimal pour le problème TSP. On note également $c(E)$ le coût associé à un chemin E, qu'il soit représenté par une suite ordonnée d'arêtes ou de sommets.

Q13. Montrer que $c(T) \leq c(H^*)$.

On ajoute un sommet dans L dès qu'on le rencontre lors du parcours en profondeur de T. Dans L, certains sommets sont donc visités plus d'une fois. La **figure 2** donne un exemple de la construction de $L = [1, 2, 3, 2, 8, 2, 1, 4, 5, 6, 5, 7, 5, 4, 1]$.



Graphe G — Étape 2 (arbre T) — Construction de L

Figure 2 - Exemple de calcul de l'étape 3 de l'**algorithme 1**

Q14. Montrer que $c(L) \leq 2c(H^*)$.

Q15. Montrer qu'il est possible de supprimer la visite d'un sommet dans L sans augmenter le coût du parcours.

Q16. En déduire comment construire H, résultat de l'**algorithme 1**.

Q17. En conclure que l'**algorithme 1** produit une 2-approximation de TSP.

Les sommets du graphe étant codés par des entiers, on donne les structures de données suivantes :

```
// Structure de graphe
struct Graphe_s {
    int n;              // Nombre de sommets
    int** C;            // Matrice des coûts
};
typedef struct Graphe_s Graphe;
```

```
// Structure Arbre Couvrant de coût minimal
struct ACM_s {
    int* parent;      // Tableau indiquant le parent de chaque sommet dans l'arbre
    int n;            // Nombre de sommets de l'arbre
};
typedef struct ACM_s ACM;

// Structure Cycle Hamiltonien
struct CH_s {
    int* cycle;       // Sommets dans l'ordre d'apparition dans le cycle
    int l;            // Longueur du cycle = taille du tableau cycle
};
typedef struct CH_s CH;
```

On admet, de plus, disposer d'une fonction de prototype `ACM* Recherche_ACM(Graphe* G)` qui calcule l'arbre couvrant de coût minimal de `G`. Cette fonction alloue la structure `ACM` :

```
ACM* a = malloc(sizeof(ACM));
a->n = n;
a->parent = malloc(n * sizeof(int));
```

avant de la calculer et de la renvoyer.

Soit un fichier texte, présent sur le disque et correctement écrit, de la forme :

fichier.txt

```
n p
s1 t1 c1
s2 t2 c2
...
sp tp cp
```

où `n` est le nombre de sommets du graphe, `p` le nombre d'arêtes, les `p` lignes suivantes donnant les deux sommets `si` et `ti` et le coût `ci` de l'arête correspondante.

Q18. Écrire une fonction de prototype `Graphe* creer_graphe(char* nom_fichier)` qui créé un graphe à partir de la lecture d'un fichier texte dont le nom est spécifié dans la chaîne de caractères `nom_fichier`. On rappelle les fonctions suivantes de gestion des fichiers :

- `FILE *fopen(char *nom_fichier, char *accessMode)` ouvre un fichier de nom `nom_fichier` selon le mode d'ouverture spécifié par `accessMode`. Pour une ouverture en mode lecture texte, on peut prendre `accessMode="r"` ;
- `int fclose(FILE *f)` ferme le fichier pointé par `f` ;
- `int fscanf(FILE *f, char *s, ...)` lit dans le fichier pointé par `f` une chaîne de caractères et utilise le paramètre `s` pour préciser le format à utiliser pour décoder la chaîne de caractères. Les valeurs lues sont stockées, une à une, dans les paramètres suivants de la fonction, dont la mémoire doit avoir été allouée. Ainsi, `fscanf(f,"%d %d",&i,&j);` lit dans `f` une ligne composée de deux entiers, séparés par un espace et les stocke aux adresses `&i` et `&j`.

Q19. Écrire une fonction de prototype `void dfs(ACM* a, int s, int* chemin, int* indice)` qui réalise un parcours en profondeur de l'arbre `a` dont la racine est `s`. `chemin` est un tableau d'entiers, `chemin[i]` contenant le i-ème sommet du parcours : à chaque fois qu'un sommet est exploré, il est ajouté au tableau `chemin` à la position spécifiée par `*indice` (passage par pointeur de l'entier).

Q20. Écrire une fonction de prototype `CH* calcule_CH(int* chemin, int n)` qui transforme le `chemin` calculé par `dfs` en un cycle hamiltonien. On prendra soin d'allouer correctement la structure de donnée retournée par la fonction et de refermer le chemin pour obtenir un cycle.

Q21. Écrire des fonctions `void free_ACM(ACM* a)` et `void free_CH(CH* cycle)` qui désallouent la mémoire précédemment allouée. On supposera que la fonction `Recherche_ACM` n'effectue pas d'allocation mémoire supplémentaire.

Q22. Écrire une fonction de prototype `CH* TSP2(Graphe* G)` qui calcule une 2-approximation de TSP. On prendra soin d'allouer/désallouer la mémoire lorsque cela est nécessaire.

Partie II - Suite de Prouhet-Thue-Morse

Cette partie comporte des questions nécessitant un code `OCaml`.

L'objectif de cette partie est d'étudier quelques définitions et propriétés de la suite de Prouhet-Thue-Morse (abrégé PTM), du nom des mathématiciens français, norvégien et américain Eugène Prouhet, Axel Thue et Marston Morse.

Notations

Dans toute la suite, on note :

- $\Sigma = \{0, 1\}$ un alphabet, Σ^* l'ensemble des mots sur Σ, ε le mot vide et $|m|$ la longueur d'un mot m,
- "." l'opérateur de concaténation de mots de Σ^*,
- si $m \in \Sigma^*$, $\bar{m}$ le mot obtenu en remplaçant dans m les 0 par de 1 et les 1 par des 0,
- pour $n \in \mathbb{N}$, t_n le n-ème terme de la suite PTM,
- pour $n \in \mathbb{N}$, $T_n = t_0 t_1 \cdots t_{2^n-1}$ le mot construit l'aide des 2^n premiers termes de la suite PTM.

II.1 - Définitions

Il existe plusieurs manières de définir la suite PTM. Nous proposons ici d'en illustrer quelques unes et de montrer leur équivalence.

Définition 1 (première définition)

Pour $n \in \mathbb{N}$, on écrit n en base 2 sous la forme $n = \sum_i d_i 2^i$ et on note $b_n = \sum_i d_i$. On définit alors $t_0 = 0$ et pour $n > 0$, $t_n = \begin{cases} 1 & \text{si } b_n \text{ impair} \\ 0 & \text{sinon} \end{cases}$.

Q23. Donner T_3.

Q24. Écrire une fonction récursive de signature `compte_bits_a_un : int -> int` telle que `compte_bits_a_un n` renvoie le nombre de bits à 1 dans l'écriture en base 2 de l'entier n.

Q25. En déduire une fonction de signature `affiche_ptm : int -> unit` qui affiche le mot T_n. On écrira une fonction auxiliaire récursive de signature `puissance2 : int -> int` qui calcule 2^n. Pour afficher les t_i, on utilisera la fonction `print_int` de signature `print_int : int -> unit`.

Définition 2 (deuxième définition)

On pose $t_0 = 0$. On suppose que pour $n > 0$ on a construit le mot T_n. On construit alors le mot T_{n+1} et donc les termes $t_{2^n}, \cdots t_{2^{n+1}-1}$ de la suite PTM par négation binaire : $\forall i \in [\![0, 2^n - 1]\!]\ t_{2^n+i} = \bar{t_i}$. Autrement dit, $T_{n+1} = T_n \bar{T}_n$.

Q26. Montrer par récurrence que cette définition calcule la même suite que la première définition.

Définition 3 (troisième définition)

On définit la suite PTM de manière récursive : $t_0 = 0$ et pour tout $n > 0$ $t_{2n} = t_n$, $t_{2n+1} = \bar{t_n}$.

Q27. Montrer que cette définition calcule la même suite que la première ou la deuxième définition.

Définition 4 (quatrième définition)
Un morphisme sur Σ^* est une fonction $\mu : \Sigma^* \to \Sigma^*$ vérifiant $\mu(\varepsilon) = \varepsilon$ et : $\forall u, v \in \Sigma^* \ \mu(u.v) = \mu(u).\mu(v)$.

On définit alors le morphisme suivant sur les éléments de Σ : $\mu(a) = \begin{cases} 01 & \text{si } a = 0 \\ 10 & \text{si } a = 1 \end{cases}$
et la suite $(u_i, i \in \mathbb{N})$ par : $u_0 = 0$ et pour tout $i \in \mathbb{N}^*$ $u_{i+1} = \mu(u_i)$.

Q28. Montrer que pour tout $i \in \mathbb{N}^*$, $u_i = T_i$.

On représente maintenant un mot de Σ^* par une liste d'entiers.

Q29. Écrire une fonction de signature `thue_morse_4 : int -> int list`, un appel à `thue_morse_4 n` générant T_n. Cette fonction fera appel à une fonction récursive de signature `morphisme : int list -> int list` renvoyant l'application du morphisme μ sur la liste d'entiers en entrée. On lèvera une exception `INVALID_ARGUMENT` si le mot donné en entrée n'est pas binaire.

II.2 - Mot infini de Thue-Morse

Le mot infini de Thue-Morse T est le mot obtenu en itérant μ une infinité de fois, en partant du symbole 0. Nous allons ici nous intéresser à la structure de T.

Définition 5 (carré, cube)
Soit $m \in \Sigma^*$. m est un carré s'il s'écrit $m = w.w$, où $w \in \Sigma^*$. C'est un cube si $m = w.w.w$, où $w \in \Sigma^*$.

Par exemple, le mot m = 110110110 est un cube avec w = 110.

Q30. Montrer qu'il n'existe pas de mots de Σ^* de longueur supérieure ou égale à 4 sans carrés.

Q31. Écrire une fonction récursive de signature `divise_mot : int list -> int list * int list` qui divise un mot m en deux sous-mots de taille identique, le premier constitué des $|m|/2$ premiers éléments de m, le second des éléments restants. On supposera ici que $|m|$ est paire.

Q32. En déduire une fonction de signature `est_carre : int list -> bool` qui détermine si un mot m est un carré. On prendra soin de vérifier que $|m|$ est paire.

Q33. Montrer que T ne contient aucun cube de la forme 000 ou 111.

Définition 6 (entrelacement)
$m \in \Sigma^*$ est un entrelacement s'il contient un facteur de la forme xyz où x, y, z sont des facteurs non vides de m et $xy = yz$. On dit que les deux facteurs xy et yz se chevauchent sur la partie commune y.

Par exemple, le mot 10101001 est un entrelacement, puisqu'il contient le facteur 010 en positions 2 et 4, qui se chevauchent sur le 0 en position 4 (en gras).

Q34. Écrire une fonction récursive de signature `sousliste : int list -> int -> int -> int list` telle que `sousliste lst i l` renvoie la sous-liste de `lst` qui commence à l'indice `i` et de longueur `l`. Si la sous-liste dépasse la fin de la liste `lst`, elle est tronquée.

Q35. Écrire une fonction de signature `est_entrelacement : int list -> bool` telle que l'appel `est_entrelacement m` renvoie `true` si `m` est un entrelacement, `false` sinon. On pourra tester récursivement toutes les longueurs de facteurs possibles et, pour un facteur donné, vérifier s'il apparaît ailleurs dans le mot avec chevauchement. On utilisera obligatoirement la fonction `sousliste`.

Q36. Montrer que si un mot m contient un facteur cube, alors c'est un entrelacement.

On admet dans la suite que si un mot est un entrelacement dont la partie commune est de longueur $k \geq 1$, alors c'est également un entrelacement de partie commune de longueur 1.

Q37. Montrer que si un mot m est un entrelacement, alors il contient un facteur de la forme $avava$, où $a \in \Sigma$ et $v \in \Sigma^*$.

Q38. Soit $m = a_0 a_1 \cdots a_{2n-1} \in \Sigma^*$ tel que, pour tout $i \in [\![0, n-1]\!]$, $a_{2i}a_{2i+1}$ s'écrit soit 01, soit 10. Montrer qu'il n'est pas possible d'écrire les mots $0m0$ et $1m1$ sous la forme $b_0 b_1 \cdots b_{2n+1}$, où, pour tout $i \in [\![0, n]\!]$, $b_{2i}b_{2i+1}$ s'écrit soit 01, soit 10.

Soit $m \in \Sigma^*$. On construit le mot $\mu(m)$ où μ est défini après la **définition 4**. Si $\mu(m) = wavavaz$ où $a \in \Sigma$ et $v, w, z \in \Sigma^*$, on peut montrer qu'alors v contient un nombre impair de symboles.

Q39. Montrer, en utilisant **Q37** et l'indication précédente, que si $\mu(m)$ est un entrelacement, alors m est un entrelacement.

Q40. Montrer par récurrence que pour tout $n \in \mathbb{N}$, T_n n'est pas un entrelacement.

Q41. En déduire que pour tout $n \in \mathbb{N}$, T_n ne contient pas de cube.

Par passage à la limite, on admet alors que T n'est pas un entrelacement et ne contient donc pas de cube.

FIN