
Épreuve d'informatique Info C — Sujet 0

Ce sujet est fourni comme exemple de sujet de l'épreuve Info C du concours X-ENS. Nous avons choisi d'y faire figurer trois exercices distincts afin de couvrir différentes parties du programme d'informatique de CPGE MP2I/MPI. Les sujets de cette épreuve proposés au concours pourront suivre un format similaire ou une organisation différente avec un unique problème.

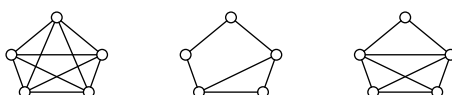
Ce sujet est composé de trois problèmes indépendants. Pour le premier problème, le langage de programmation est C. Pour les deux autres problèmes, le langage de programmation est OCaml.

Partie I. Circuits eulériens

Ce problème utilise le langage C.

Dans ce problème, on s'intéresse uniquement à des graphes *non orientés*, contenant au moins un sommet et *connexes*, c'est-à-dire où il existe toujours un chemin entre deux sommets donnés. On supposera également qu'il n'y a pas de boucle c'est à dire qu'il n'existe pas d'arête ayant pour source et destination un même sommet. Dans ce contexte, un *cycle eulérien* est un chemin qui part d'un sommet et revient à ce dernier et qui emprunte *chaque arête une et une seule fois*.

Question 1. Pour chacun des trois graphes suivants, indiquer s'il possède ou non un cycle eulérien. Dans le cas positif, en exhiber un. On ne demande pas de justification.



Question 2. Montrer que si un graphe admet un cycle eulérien alors tous ses sommets sont de degré pair. On rappelle que le *degré* d'un sommet est le nombre de ses voisins.

Réciproque. Dans la suite de ce problème, on va démontrer que la réciproque est vraie, c'est-à-dire qu'un graphe dont tous les sommets sont de degré pair admet un cycle eulérien. On va le faire en programmant un algorithme qui construit un cycle eulérien. On identifie les sommets avec des entiers.

Pour construire un cycle eulérien, on va représenter toute arête $a - b$ du graphe par deux arcs orientés inverses ($a \rightarrow b$ et $b \rightarrow a$) en utilisant la structure Edge donnée figure 1. Une valeur de

type `Edge` représente un arc du graphe entre les sommets `src` et `dst`. Les deux autres champs `prev` et `next` permettent de chaîner les arcs dans une liste circulaire doublement chaînée, afin de représenter un cycle dans le graphe. Même si le graphe est non orienté, on choisit donc de voir une structure `Edge` comme allant de `src` à `dst`, de telle sorte que si deux valeurs a et b de type `Edge` sont liées, dans cet ordre, on aura

```
a.next = &b
b.prev = &a
a.dst = b.src
```

Une fonction `connect` est fournie pour réaliser une telle connection. Une autre fonction, `join`, est fournie et sera utilisée plus loin.

Le graphe est décrit figure 2. On identifie les sommets avec les entiers $0, 1, \dots, N - 1$. Le graphe est donné dans une variable globale `graph`, sous la forme d'une matrice d'adjacence, dans laquelle on représente toute arête du graphe non orienté par deux arcs orientés inverses. S'il n'y a pas d'arc entre les sommets i et j , alors `graph[i][j]` et `graph[j][i]` valent `NULL`. S'il y a un arc entre les sommets i et j , alors `graph[i][j]` pointe vers une structure e telle que $e.\text{src} = i$ et $e.\text{dst} = j$ et, inversement, `graph[j][i]` pointe vers une structure e telle que $e.\text{src} = j$ et $e.\text{dst} = i$. Les champs `next` et `prev` de ces deux structures sont `NULL`. Autrement dit, les arcs ne sont pas connectés entre eux initialement. Au fur et à mesure de la construction du cycle eulérien, ces arcs seront retirés du graphe et connectés entre eux pour former un cycle.

Question 3. On commence par un simple test que le critère est vérifié. Écrire une fonction

```
bool is_eulerian()
```

qui détermine si le graphe possède uniquement des sommets de degré pair. Votre fonction devra laisser le graphe inchangé.

Deux fonctions sont fournies. La fonction `has_edges` détermine si un sommet donné possède au moins un arc. Étant donné un sommet v qui possède au moins un arc, la fonction `any_edge_from` renvoie un arc e tel que $e.\text{src} = v$, après l'avoir supprimé du graphe (ainsi que l'arc inverse).

Question 4. Soit un graphe possédant au moins deux sommets dont tous les sommets ont un degré pair. Soit un sommet v_0 dans ce graphe. Montrer que le graphe possède un cycle (pas nécessairement eulérien) partant de v_0 . On rappelle que le graphe est connexe.

Question 5. On suppose que le graphe contient encore des arcs sortant d'un sommet `start` et que tous les sommets sont de degré pair. Écrire une fonction

```
edge *round_trip(int start)
```

qui construit un cycle à partir du sommet `start` en utilisant des arcs encore présents dans le graphe et les retire au fur et à mesure (grâce à la fonction `any_edge_from`). L'arc renvoyé est l'arc de ce cycle dont le champ `src` est `start`. Cette fonction ne pourra pas échouer, en vertu de la question précédente. On prendra soin de bien mettre à jour les champs `next` et `prev` des arcs utilisés, en appelant la fonction `connect` fournie.

```

typedef struct Edge {
    int src, dst;
    struct Edge *next, *prev;
} edge;

// relie l'arc x à l'arc y, en supposant x->dst == y->src
void connect(edge *x, edge *y) {
    assert(x->dst == y->src);
    x->next = y;
    y->prev = x;
}

// joint les deux cycles c1 et c2, en supposant c1->src == c2->src
void join(edge *c1, edge *c2) {
    edge *p = c1->prev;
    assert(c1->src == c2->src);
    edge *q = c2->prev;
    connect(p, c2);
    connect(q, c1);
}

```

FIGURE 1 – Une structure pour représenter les chemins.

Question 6. Écrire une fonction

```
edge *find_vertex_with_edges(edge *c)
```

qui reçoit en argument un cycle et cherche le long de ce cycle un sommet qui possède encore des arcs. S'il existe un tel sommet v , cette fonction renvoie un arc e de ce cycle pour lequel $e.\text{src} = v$. Si en revanche il n'existe aucun sommet le long du cycle possédant encore des arcs, cette fonction renvoie `NULL`.

Algorithme de Hierholzer. Pour construire un cycle eulérien, on peut procéder ainsi. On commence par construire un cycle arbitraire, avec la fonction `round_trip` et en partant d'un sommet initial quelconque. Si tous les arcs ont été utilisés, on a terminé. Sinon, on prend un sommet v sur le cycle qui possède encore des arcs. C'est possible, car le graphe est connexe. À partir de v , on construit un nouveau cycle, toujours avec la fonction `round_trip`. Puis on joint les deux cycles pour n'en former qu'un seul. Et ainsi de suite jusqu'à épuisement des arcs. Pour joindre deux cycles, on utilise la fonction `join` fournie dans la figure 1.

Question 7. Écrire une fonction

```
edge *eulerian_cycle(int start)
```

qui reçoit en argument un sommet initial `start` et construit et renvoie un cycle eulérien en suivant l'algorithme de Hierholzer.

Question 8. Donner la complexité dans le pire des cas de l'algorithme de Hierholzer tel que nous l'avons écrit, en fonction du nombre N de sommets et du nombre E d'arcs du graphe. Est-il

```
#define N ...
    // la taille du graphe

edge *graph[N][N];
    // et sa matrice d'adjacence

bool has_edges(int v) {
    // indique s'il y a au moins un arc sortant de v
    for (int j = 0; j < N; j++)
        if (graph[v][j] != NULL)
            return true;
    return false;
}

edge *any_edge_from(int v) {
    // renvoie un arc sortant de v
    // l'arc est supprimé du graphe (et l'arc inverse également)
    for (int j = 0; j < N; j++) {
        if (graph[v][j] == NULL) continue;
        edge *e = graph[v][j];
        graph[v][j] = NULL;
        graph[j][v] = NULL;
        return e;
    }
    return NULL;
}
```

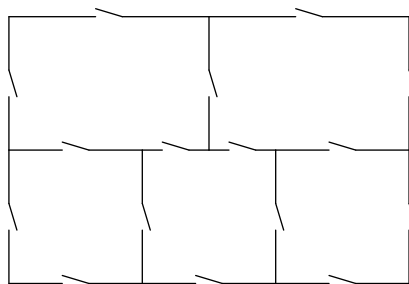
FIGURE 2 – Une structure pour représenter le graphe.

possible d'obtenir une meilleure complexité pour la construction d'un cycle eulérien ? (Si oui, on ne demande pas d'écrire le code, mais seulement d'en décrire les idées.)

Chemin eulérien. Un *chemin* eulérien dans un graphe non orienté est un chemin qui emprunte chaque arête une et une seule fois, sans nécessairement revenir au point de départ.

Question 9. Montrer qu'un graphe admet un chemin eulérien si et seulement s'il possède 0 ou 2 sommets de degré impair. Pour la réciproque, on s'attachera à décrire un algorithme construisant le chemin (mais on ne demande pas d'en écrire le code).

Question 10. Est-il possible de tracer un chemin continu qui traverse les cinq pièces suivantes en empruntant chaque porte une et une seule fois ?



Le chemin n'a pas besoin d'être cyclique. Si oui, exhiber un tel chemin ; si non, justifier.

Partie II. Analyse descendante

Ce problème utilise le langage OCaml.

Dans ce problème, on étudie un algorithme d'analyse syntaxique, appelé analyse descendante, qui s'applique à certaines grammaires non contextuelles. Pour une grammaire donnée, on note N l'ensemble de ses non terminaux (notés avec des majuscules) et T l'ensemble de ses terminaux (notés avec des minuscules). Les lettres grecques (α, β, γ , etc.) désignent des mots de $(N \cup T)^*$. Le mot vide est noté ε . Une règle de production de la grammaire est notée $X \rightarrow \gamma$. Une dérivation immédiate est notée $\alpha \Rightarrow \beta$, ce qui signifie qu'il existe une règle de production $X \rightarrow \gamma$ avec $\alpha = \alpha_1 X \alpha_2$ et $\beta = \alpha_1 \gamma \alpha_2$. On note \Rightarrow^* la clôture réflexive transitive de la relation \Rightarrow , c'est-à-dire une dérivation en un nombre quelconque d'étapes, y compris zéro.

On prend en exemple une version simplifiée de la grammaire du langage de programmation LISP, avec un ensemble de quatre terminaux $T = \{\text{sym}, (,), \#\}$, un ensemble de trois non terminaux $N = \{S, L, E\}$, dont le symbole initial S , et les cinq règles de production suivantes :

$$\begin{array}{lcl} S & \rightarrow & L \# \\ L & \rightarrow & \varepsilon \\ & | & E L \\ E & \rightarrow & \text{sym} \\ & | & (L) \end{array}$$

Appelons G cette grammaire.

Question 11. Pour un entier $n \in \mathbb{N}$ arbitraire, donner un mot de longueur au moins n engendré par cette grammaire et la dérivation à gauche correspondante.

Pour réaliser l'analyse syntaxique de ce petit langage avec OCaml, on se donne un type `token` pour les symboles terminaux `sym`, `(`, `)` et `#` :

```
type token = Sym | Lpar | Rpar | Eof
```

Ainsi `Sym` représente le terminal `sym`, `Lpar` le terminal `(`, `Rpar` le terminal `)` et `Eof` le terminal `#`.

Notre objectif est d'écrire une fonction `accepts: token list -> bool` qui détermine si un mot, donné comme une liste de terminaux, appartient ou non au langage de cette grammaire. Pour cela, on va commencer par construire trois fonctions, une pour chaque non terminal de la grammaire, avec les types suivants :

```
val parseS: token list -> token list
val parseL: token list -> token list
val parseE: token list -> token list
```

La fonction `parseX` reconnaît un préfixe maximal de la liste passée en argument comme étant un mot dérivé de X et renvoie le reste de la liste. S'il n'y a pas de tel préfixe, alors la fonction lève l'exception `SyntaxError`, que l'on suppose définie.

Pour définir ces trois fonctions, on se donne une table à deux entrées appelée *table LL*. Cette table indique, pour un non terminal que l'on cherche à reconnaître et pour un terminal au début de la liste, la règle de production à utiliser. Voici cette table pour notre grammaire :

	sym	()	#
<i>S</i>	<i>L</i> #	<i>L</i> #		<i>L</i> #
<i>L</i>	<i>EL</i>	<i>EL</i>	ε	ε
<i>E</i>	sym	(<i>L</i>)		

La fonction `parseS` est donc définie en suivant la première ligne de cette table et voici son code :

```
let rec parseS l = match l with
| (Sym | Lpar | Eof) :: _ ->
    (match parseL l with Eof :: q -> q | _ -> raise SyntaxError)
| [] | Rpar :: _ -> raise SyntaxError
```

En particulier, une case vide dans la table est interprétée comme un échec de l'analyse.

Question 12. Donner le code des fonctions `parseL` et `parseE`.

Question 13. Donner le code de la fonction `accepts`.

Construction de la table. On va maintenant chercher à construire une telle table pour une grammaire arbitraire. On introduit pour cela une première notion : On dit qu'un non terminal X est *nul*, et on note $\text{NUL}(X)$, si le mot vide peut être dérivé de X , c'est-à-dire $X \Rightarrow^* \varepsilon$.

Question 14. Indiquer quels sont les symboles nuls de la grammaire G prise en exemple plus haut.

Calcul des symboles nuls. Pour déterminer $\text{NUL}(X)$, on propose l'algorithme suivant.

1. Initialement, on fixe $\text{NUL}(X)$ à **false** pour tout $X \in N$.
2. Pour chaque non terminal X , on affecte la valeur **true** à $\text{NUL}(X)$ s'il existe une production $X \rightarrow \varepsilon$ ou une production $X \rightarrow X_1 X_2 \dots X_p$ avec X_i des non terminaux et $\text{NUL}(X_i)$ pour tout i .
3. Si l'étape 2 a modifié au moins une valeur $\text{NUL}(X)$, alors on recommence l'étape 2.

Question 15. Montrer que cet algorithme termine.

Question 16. Montrer que cet algorithme détermine bien la valeur de $\text{NUL}(X)$.

Les premiers et les suivants. On introduit deux autres notions sur la grammaire. Pour un non terminal $X \in N$, on définit deux ensembles de terminaux :

- $\text{PREMIERS}(X)$ est l'ensemble des terminaux qui peuvent apparaître au début des mots dérivés depuis X , c'est-à-dire $\text{PREMIERS}(X) = \{t \in T \mid \exists \alpha \in (N \cup T)^*. X \Rightarrow^* t\alpha\}$;

- $\text{SUIVANTS}(X)$ est l'ensemble des terminaux qui peuvent apparaître après X dans une dérivation, c'est-à-dire $\text{SUIVANTS}(X) = \{t \in T \mid \exists \alpha, \beta \in (N \cup T)^*. S \Rightarrow^* \alpha X t \beta\}$.

Question 17. Donner les ensembles $\text{PREMIERS}(X)$ et $\text{SUIVANTS}(X)$ pour la grammaire prise en exemple plus haut (soit six ensembles au total).

Construction de la table. On admet que l'on peut calculer les ensembles $\text{PREMIERS}(X)$ et $\text{SUIVANTS}(X)$ pour toute grammaire. On étend NUL et PREMIERS sur tout mot de $(N \cup T)^*$ de la manière suivante :

$$\begin{aligned} \text{NUL}(\varepsilon) &= \text{true} \\ \text{NUL}(X_1 X_2 \dots X_p) &= \text{true} \quad \text{si } \text{NUL}(X_i) = \text{true} \text{ pour tout } i \\ \text{PREMIERS}(\varepsilon) &= \emptyset \\ \text{PREMIERS}(t\alpha) &= \{t\} \\ \text{PREMIERS}(X\alpha) &= \text{PREMIERS}(X) \quad \text{si } \text{NUL}(X) = \text{false} \\ \text{PREMIERS}(X\alpha) &= \text{PREMIERS}(X) \cup \text{PREMIERS}(\alpha) \quad \text{si } \text{NUL}(X) = \text{true} \end{aligned}$$

où t est un terminal et X un non terminal.

On construit alors la table LL de la manière suivante : pour un non terminal $X \in N$ et un terminal $t \in T$, on indique la règle de production $X \rightarrow \gamma$ dans la case (X, t) de la table

- si $t \in \text{PREMIERS}(\gamma)$,
- ou si $\text{NUL}(\gamma)$ et $t \in \text{SUIVANTS}(X)$.

On peut alors se servir de cette table pour réaliser une analyse syntaxique dès lors que chaque case ne comporte qu'au plus une règle de production. *On pourra vérifier que l'on obtient bien la table donnée au début de cette partie avec les résultats obtenus aux questions 14 et 17.*

Un autre exemple. On considère une seconde grammaire G' , sur les mêmes symboles terminaux, avec des non terminaux $\{S', L', E'\}$ et un symbole initial S' :

$$\begin{array}{lcl} S' & \rightarrow & L' \# \\ L' & \rightarrow & \varepsilon \\ & & | \quad L' E' \\ E' & \rightarrow & \text{sym} \\ & & | \quad (L') \end{array}$$

Question 18. Montrer que G' reconnaît le même langage que G .

Question 19. Construire la table LL pour cette seconde grammaire. Permet-elle de coder un algorithme pour l'analyse syntaxique des mots générés par cette grammaire ?

Partie III. Logique propositionnelle

Ce problème utilise le langage OCaml.

Dans ce problème, on considère des formules de logique propositionnelle sur n variables notées X_i avec $0 \leq i < n$. Une valeur de vérité est un élément de $\mathbb{B} = \{V, F\}$. Une valuation est une fonction v de $\{0, 1, \dots, n-1\}$ dans \mathbb{B} , qui assigne une valeur de vérité à chacune des variables. On note $\mathcal{V}_v(f)$ la valeur de vérité de la formule f pour la valuation v .

On choisit de représenter nos formules en OCaml avec le type suivant.

```
type formula =  
  | True  
  | False  
  | If of int * formula * formula
```

La valeur de vérité d'une telle formule est définie de la manière suivante :

$$\begin{aligned}\mathcal{V}_v(\text{True}) &= V \\ \mathcal{V}_v(\text{False}) &= F \\ \mathcal{V}_v(\text{If}(i, f, g)) &= \text{si } v(i) = V \text{ alors } \mathcal{V}_v(f) \text{ sinon } \mathcal{V}_v(g)\end{aligned}$$

On ajoute par ailleurs la contrainte que toute formule est *ordonnée*, au sens où si elle est de la forme $\text{If}(i, f, g)$, alors les formules f et g ne font intervenir que des variables *strictement plus grandes que* i . Ainsi,

$\text{If}(0, \text{If}(1, \text{False}, \text{True}), \text{If}(2, \text{True}, \text{False}))$

est une formule ordonnée, mais

$\text{If}(0, \text{If}(1, \text{False}, \text{True}), \text{If}(0, \text{True}, \text{False}))$

n'en est pas une.

Question 20. Écrire une fonction `check: int -> formula -> bool` qui prend en arguments un entier n et une formule f et qui détermine si d'une part f est bien une formule ordonnée et si d'autre part f est limitée aux variables X_i avec $0 \leq i < n$. La complexité doit être linéaire en la taille de la formule. On ne demande pas de justifier la complexité.

Question 21. Donner une formule ordonnée pour $n = 3$ variables qui est vraie si et seulement si les trois variables ont la même valeur de vérité.

Tautologies. Afin de décider si une formule est une tautologie, on se donne le type OCaml `result` suivant :

```
type assignment = bool array  
type result = Tautology | Refutation of assignment
```

Le type `assignment` correspond à une valuation. Une valeur de type `assignment` est un tableau `a` de taille n , où `a.(i)` donne la valeur de la variable X_i .

Question 22. Écrire une fonction `decide: int -> formula -> result` qui prend en arguments un entier n et une formule f , supposée ordonnée et sur n variables, et qui renvoie

- `Tautology` si f est une tautologie;
- `Refutation v` sinon, avec $\mathcal{V}_v(f) = F$.

On s'efforcera de proposer quelque chose de plus efficace que le test systématique de toutes les valuations possibles, en faisant intervenir le caractère ordonné de la formule.

Construire des formules arbitraires. Pour montrer que toute formule booléenne classique admet une formule ordonnée équivalente, il suffit de se donner des fonctions sur les formules ordonnées qui correspondent aux connecteurs logiques usuels, telles que la négation, la conjonction, la disjonction, etc.

Question 23. Écrire une fonction `mk_not: formula -> formula` qui reçoit en argument une formule ordonnée f et qui renvoie une formule ordonnée correspondant à la négation $\neg f$, c'est-à-dire une formule ordonnée g telle que, pour toute valuation v , on a $\mathcal{V}_v(g) = \neg \mathcal{V}_v(f)$. Donner, sans la justifier, la complexité de votre code en fonction de la taille de f .

Question 24. Écrire une fonction `mk_or: formula -> formula -> formula` qui reçoit en arguments deux formules ordonnées f et g et qui renvoie une formule ordonnée correspondant à la disjonction $f \vee g$, c'est-à-dire une formule ordonnée h telle que, pour toute valuation v , on a $\mathcal{V}_v(h) = \mathcal{V}_v(f) \vee \mathcal{V}_v(g)$. Donner, sans la justifier, la complexité de votre code en fonction de la taille des formules f et g .

Poisson d'avril. Le poisson d'avril est une espèce extrêmement rare qui n'a jamais été permis d'observer dans la nature. Néanmoins, les ichtyologues ont permis de déterminer les faits suivants :

- (F_1) tout poisson d'avril qui ne nage pas en mer chaude a des rayures rouges ;
- (F_2) tout poisson d'avril a des nageoires bleues ou n'a pas de rayures rouges ;
- (F_3) les poissons d'avril qui vivent dans le corail ne mangent pas de crevettes ;
- (F_4) un poisson d'avril mange des crevettes si et seulement s'il nage en mer chaude ;
- (F_5) tout poisson d'avril qui a des nageoires bleues nage en mer chaude et vit dans le corail ;
- (F_6) tout poisson d'avril qui nage en mer chaude a des nageoires bleues.

On souhaite mettre en application le code OCaml développé plus haut pour démontrer qu'il n'existe pas de poisson d'avril.

Question 25. Expliquer comment construire une formule ordonnée de type `formula` qui est une tautologie si et seulement si les poissons d'avril n'existent pas.

* *

*