DS n° 03 — 3h

On veillera à présenter *très clairement* sa copie, on attachera un soin particulier à la rédaction et **on encadrera les réponses**. Des points pourront être accordés à la présentation, à la lisibilité et au soin accordé à la rédaction. Il est impératif d'utiliser un brouillon. Les programmes seront rédigés clairement et proprement, en mettant en valeur l'indentation et en choisissant des noms de variables explicites. **Les commentaires des programmes doivent dans tous les cas être dans une autre couleur que le programme lui-même**. Il est impératif de respecter l'indentation usuelle des fonctions OCaml et C.

Lorsque l'on pose des hypothèses sur les arguments, il n'est pas nécessaire dans vos fonctions de le vérifier ni de prévoir le comportement si l'hypothèse est violée. L'introduction et l'utilisation de fonctions auxiliaires est autorisée, et même encouragée, d'autant plus si cela améliore la lisibilité et la compréhension. Les fonctions auxiliaires **doivent être précédées de leur type (en OCaml) et commentées**.

Le sujet est composé d'une première partie de questions de cours qu'il faut impérativement traiter en premier, puis de trois parties indépendantes, de difficulté croissante, mais que vous pouvez traiter dans l'ordre de votre choix.

Questions de cours

Échauffement

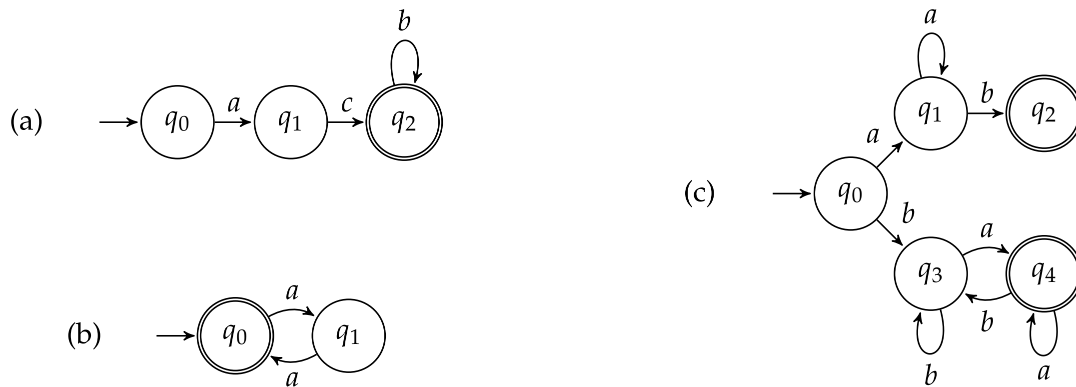
i) Donner la ligne de compilation en C que vous devez désormais systématiquement utiliser.
ii) Donner la définition formelle d'un automate déterministe fini.
iii) Quel est l'invariant qui permet de justifier la correction de l'algorithme de Kruskal ?

Déduction naturelle

iv) Démontrer que les séquents suivants sont dérivables.
   (a) En logique minimale.
   $$A \land B, B \land C \vdash A \land C$$
   (b) En logique minimale. On pourra noter $\Gamma = \{A \to (B \lor C), \neg B, \neg C, A\}$.
   $$A \to (B \lor C), \neg B, \neg C \vdash \neg A$$
   (c) En logique intuitionniste. On pourra noter $\Gamma = \{(A \lor C) \to (B \lor C), \neg C, A\}$.
   $$(A \lor C) \to (B \lor C), \neg C \vdash A \to B$$
   (d) En logique classique. On pourra noter $\Gamma = \{C \to B, \neg C \to \neg A, A\}$.
   $$C \to B, \neg C \to \neg A \vdash A \to B$$
v) Les séquents suivants sont-ils dérivables (en logique classique) ?
   (a) $\neg(A \to B) \vdash B \to A$.
   (b) $\neg(A \to B) \vdash A \to B$.

Automates

vi) Soit $\mathcal{A}$ un automate sur Σ, d'états Q et de fonction de transition δ. Montrer *très soigneusement* que pour un état $q \in Q$ et deux mots $u, v \in \Sigma^*$ on a $\delta^\star(q, uv) = \delta^\star(\delta^\star(q, u), v)$.
vii) Pour chacun des automates suivants, donner une expression régulière décrivant le langage qu'il reconnaît.



viii) Dessiner des automates fini déterministes reconnaissant les langages suivants sur $\Sigma = \{a, b\}$:
   (a) L décrit par $(a|b)^*$;
   (b) $L = \{u \in \Sigma^*, \ |u|_a \pmod 2 = |u|_b \pmod 3\}$;
   (c) L décrit par $(a|b)^* a^2 (a|b)^*$.

Composantes fortement connexes

On représente des graphes (orientés ou non) en OCaml par listes d'adjacence et on suppose disposer des primitives ci-dessous. On représente les composantes (connexes ou fortement connexes) par un tableau de type `int array` associant à chaque sommet un identifiant de composante, indexées à partir de 0.

OCaml
```
type graphe = int list array
type composantes = int array

let nb_sommets g = Array.length g
let voisins g s = g.(s)
```

ix) Écrire une fonction `miroir : graphe -> graphe` qui renvoie un nouveau graphe correspondant au miroir d'un graphe orienté passé en paramètre.

x) Écrire une fonction `postfixe : graphe -> int list` qui renvoie la liste des sommets du graphe dans l'ordre inverse d'un parcours en profondeur postfixe du graphe. Le dernier sommet postvisité est donc en tête de cette liste. On pourra utiliser une référence de liste.

xi) La fonction `composantes_connexes : graphe -> int list -> composantes` calcule les composantes connexe d'un graphe (vu comme un graphe non orienté) en utilisant l'ordre donnée par une liste de sommets passée en paramètre lors de l'exploration de nouvelles composantes. Implémenter cette fonction.

xii) En déduire une fonction `kosaraju : graphe -> composantes` qui renvoie les composantes fortements connexes d'un graphe orienté.

Partie I. Logique et calcul des propositions

Nous nous intéressons dans cet exercice à l'étude de quelques propriétés de la logique propositionnelle tri-valuée. En plus des deux valeurs classiques VRAI ($\top$) et FAUX ($\bot$) que peut prendre une expression, la logique propositionnelle tri-valuée introduit une troisième valeur INDETERMINE (?).
$\mathcal{V}$ est l'ensemble des variables propositionnelles et $\mathcal{F}$ l'ensemble des formules construites sur $\mathcal{V}$. Pour $A, B \in \mathcal{V}$, les tables de vérité des opérateurs classiques dans cette logique propositionnelle sont les suivantes :

(a) $A \wedge B$

A	B	$A \wedge B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	?	?
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$
$\bot$	?	$\bot$
?	$\top$	?
?	$\bot$	$\bot$
?	?	?

(b) $A \vee B$

A	B	$A \vee B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	?	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$
$\bot$	?	?
?	$\top$	$\top$
?	$\bot$	?
?	?	?

(c) $A \Rightarrow B$

A	B	$A \Rightarrow B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	?	?
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$
$\bot$	?	$\top$
?	$\top$	$\top$
?	$\bot$	?
?	?	$\top$

(d) $\neg A$

A	$\neg A$
$\top$	$\bot$
$\bot$	$\top$
?	?

Définitions

Définition 1 Tri-valuation.
Une tri-valuation est une fonction $f : \mathcal{V} \to \{\top, \bot, ?\}$.

On étend alors de manière usuelle la notion de tri-valuation sur l'ensemble des formules :

Définition 2 Une tri-valuation sur l'ensemble des formules est une fonction $\hat{f} : \mathcal{F} \to \{\top, \bot, ?\}$.

Définition 3 Une tri-valuation $\hat{f}$ satisfait une formule ϕ si $\hat{f}(\phi) = \top$. On notera alors $\hat{f} \vdash_3 \phi$.

Définition 4 Formule.
Une formule ϕ est :
— une conséquence d'un ensemble de formules $\mathcal{X}$ si toute interprétation qui satisfait toutes les formules de $\mathcal{X}$ satisfait ϕ. On notera dans ce cas $\mathcal{X} \Vdash_3 \phi$;
— une tautologie si pour toute tri-valuation $\hat{f}$, $\hat{f}(\phi) = \top$. On notera dans ce cas $\Vdash_3 \phi$.

Questions

I.1. Montrer que $A \vee \neg A$ n'est pas une tautologie.

I.2. Proposer alors une tautologie simple dans cette logique.

Posons $\top = 1, \bot = 0$ et $? = 0,5$.

I.3. Proposer un calcul simple permettant de trouver la table de vérité de $A \wedge B$ en fonction de A et B. Même question pour $A \vee B$.

I.4. En logique bi-valuée classique, les propositions $\neg A \vee B$ et $A \Rightarrow B$ sont équivalentes. Qu'en est-il dans le cadre de la logique propositionelle tri-valuée ?

I.5. En écrivant les tables de vérité, indiquer si les propositions $\neg B \Rightarrow \neg A$ et $A \Rightarrow B$ sont équivalentes.

I.6. Donner la table de vérité de la proposition $((A \Rightarrow B) \wedge ((\neg A) \Rightarrow B)) \Rightarrow B$. Cette proposition est-elle une tautologie ?

Un nouvel opérateur d'implication, noté $\rightarrow$, est alors défini, dont la table de vérité est la suivante :

A	B	$A \rightarrow B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	?	?
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$
$\bot$	?	$\top$
?	$\top$	$\top$
?	$\bot$	?
?	?	?

I.7. $A \rightarrow A$ est-elle une tautologie ?

I.8. Montrer qu'il n'existe aucune tautologie en utilisant uniquement cette définition de l'implication.

I.9. La proposition suivante est-elle une tautologie :

" $(\{A\} \Vdash_3 B)$ est équivalent à $(\Vdash_3 A \rightarrow B)$ " ?

On définit alors un type `FormuleLogique` représentant les formules de la manière suivante : `type FormuleLogique`

```
|Vrai (* Constante Vrai*)
|Faux (* Constante Faux*)
|Indétermine (* Constante Indeterminé*)
|Var of string (* Variable propositionnelle*)
|Non of FormuleLogique (* Négation d'une formule*)
|Et of FormuleLogique*FormuleLogique (* conjonction de deux formules*)
|Ou of FormuleLogique*FormuleLogique (* disjonction de deux formules*)
|Implique of FormuleLogique*FormuleLogique (* implication*)
```

I.10. Avec la représentation précédente, écrire en CaML la formule :

$$((A \Rightarrow B) \wedge ((\neg A) \Rightarrow B)) \Rightarrow B\ .$$

I.11. Écrire alors une fonction récursive CaML `lectureFormule`, prenant en argument une formule et renvoyant une chaîne de caractères spécifiant comment un lecteur lirait la formule. Ainsi, par exemple, pour $\phi = A \wedge (\neg B)$, `lectureFormule` ϕ renvoie `A et non B`.

Partie II. Arbres couvrants et algorithme de Borůvka

Représentation d'un graphe en C

On représente un graphe non orienté pondéré $G = (S, A, p)$, avec $S = [\![0, n-1]\!]$ par le type :

```c
struct graphe_s {
  int n;
  int d;
  int** adj;
};
typedef struct graphe_s graphe;
```

Si `g` est un objet de type `graphe` correspondant à un graphe pondéré $G = (S, A, p)$ de degré d, alors `g.n` est égal à $n = |S|$, `g.d` est le degré d du graphe et si $s \in S$ est un sommet de degré k, alors `g.adj[s]` est un tableau de taille $2d+1$ tel que si on note $\{t_0, \ldots, t_{k-1}\}$ les voisins de s, alors :

- pour $i \in [\![0, k-1]\!]$, `g.adj[s][2 * i]` est égal à t_i et `g.adj[s][2 * i + 1]` est égal au poids $p(\{s, t_i\})$ de l'arête $\{s, t_i\}$;
- `g.adj[s][2 * k]` est une valeur sentinelle égale à -1 ;
- pour $i \in [\![2k+1, 2d]\!]$ `g.adj[s][i]` a une valeur quelconque.

Q 12) On considère cette représentation de graphe.

(a) Écrire une fonction `int degre(graphe g, int s)` qui détermine le degré d'un sommet s dans un graphe G.

(b) Écrire une fonction `int poids_graphe(graphe g)` qui calcule et renvoie le poids d'un graphe, c'est-à-dire la somme des poids de ses arêtes. *Attention à ne pas compter deux fois une même arête.*

(c) Écrire une fonction `void liberer_graphe(graphe g)` qui libère la mémoire utilisée par un graphe `g`. *Attention,* `g` *n'est pas un pointeur.*

Arbre couvrant de poids minimal

Dans toute la suite du problème, on suppose pour simplifier que tous les poids des arêtes sont deux à deux distincts. On admet l'unicité de l'arbre couvrant de poids minimal dans ce cas. On considère le graphe suivant à gauche et un de ses sous-graphes acyclique à droite.

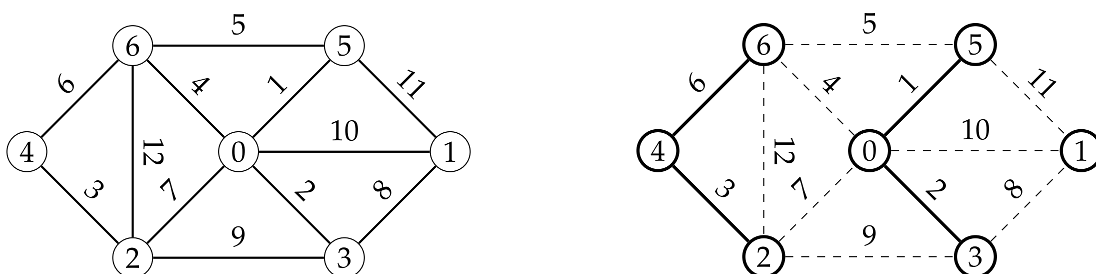


FIGURE 1 – Un graphe G (à gauche) et un sous-graphe acyclique H de G (à droite).

Q 13) On considère dans un premier temps l'algorithme de Kruskal.

(a) Rappeler très rapidement le principe de cet algorithme.

(b) Appliquer cet algorithme au graphe G proposé ci-dessus à gauche dans la figure 1 et donner l'arbre couvrant de poids minimal obtenu.

(c) Déterminer la complexité de l'algorithme de Kruskal.

Algorithme de Borůvka

Soit $G = (S, A, p)$ un graphe non orienté pondéré. On suppose que $H = (S, B)$ est un sous-graphe acyclique de G (c'est-à-dire que $B \subseteq A$). On considère $C \subseteq S$ une composante connexe de H. On appelle :

- arête **inutile** pour H une arête de $A \setminus B$ dont les deux extrémités sont dans C ;
- arête **sûre** pour H une arête de $A \setminus B$ dont le poids est minimal parmi les arêtes qui ont exactement une extrémité dans C.

Par exemple, pour le sous-graphe H de la figure 1 et pour la composante connexe $\{2, 4, 6\}$ l'arête de poids 12 est inutile et l'arête de poids 4 est sûre (minimal parmi les arêtes de poids 4, 5, 7 et 9 qui sont celles ayant exactement une extrémité dans cette composante. Remarquons qu'une arête sûre n'est donc jamais inutile car exactement une de ses extrémités est dans C. Sauf si H est connexe, chaque composante connexe de H comporte exactement une arête sûre. Certaines arêtes ne sont ni sûres ni inutiles. Une même arête peut-être sûre pour deux composantes connexes C et C'. Dans l'exemple ci-dessus, l'arête de poids 4 est également l'arête sûre pour la composante connexe $\{0, 3, 5\}$.

Q 14) Déterminer l'arête sûre pour la composantes connexe $\{1\}$ du sous-graphe H de la figure 1.

Q 15) Justifier que l'algorithme de Kruskal ignore les arêtes inutiles et ajoute à chaque étape l'arête sûre de poids minimal parmi toutes les arêtes sûres.

Q 16) On suppose dans cette question que $H = (S, B)$ est un sous-graphe acyclique de l'arbre couvrant de G de poids minimal $T^* = (S, B^*)$, c'est-à-dire que $B \subseteq B^*$. On attend dans cette question des preuves directes, n'utilisant pas l'algorithme de Kruskal et sa correction.

(a) Montrer que B^* ne contient aucune arête inutile pour H.

(b) Montrer rigoureusement que B^* contient toutes les arêtes sûres pour H.

Q 17) Écrire une fonction `int* composantes(graphe h, int* m)` qui prend en argument un graphe $H = (S, B, p)$ d'ordre n et qui renvoie un tableau `comp` de taille n tel que pour tout sommet $s \in S$, `comp[s]` est le numéro de la composante connexe de s. On supposera que les composantes connexes sont numérotées de 0 à $m - 1$. La fonction devra également modifier l'entier pointé par `m` pour y stocker la valeur de m correspondant au nombre de composantes connexes. *Indication : il n'est pas du tout interdit d'écrire une fonction auxiliaire récursive* `void explorer(graphe h, int s, int* comp, int id)`.

Q 18) Expliquer comment obtenir en $\mathcal{O}(|S| + |A|)$ l'ensemble des arêtes sûres pour un sous-graphe acyclique H. On détaillera l'algorithme proposé et les structures de données choisies mais on ne demande pas de l'implémenter en C. Justifier que la complexité obtenue est bien celle demandée.

L'algorithme de Borůvka utilise les propriétés sur les arêtes sûres pour construire un arbre couvrant de poids minimal. Il se résume de la manière suivante :

> **Entrée :** Graphe pondéré non orienté connexe $G = (S, A, p)$.
> **Début algorithme**
> - Poser $B = \varnothing$
> - **Tant que** il reste des arêtes sûres pour $H = (S, B)$ **Faire**
>   - Ajouter à B toutes les arêtes sûres pour H.
> - **Renvoyer** (S, B)

Q 19) On considère l'algorithme de de Borůvka ci-dessus.

(a) Donner le déroulé de l'algorithme sur le graphe G de la figure 1. On détaillera bien chaque étape en indiquant à chaque étape, pour chaque composantes connexe chaque étape son arête sûre et on indiquera l'ensemble des arêtes sûres ajoutées à chaque étape.

(b) Montrer que l'algorithme de Borůvka termine.

(c) Montrer que l'algorithme de Borůvka renvoie un arbre couvrant de poids minimal de G. *Indication : on utilisera le même invariant que pour montrer la correction de l'algorithme de Kruskal.*

(d) Déterminer soigneusement la complexité temporelle de cet algorithme.

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 $\{0, 1, \ldots, n-1\}$ 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 :

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

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

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

est une formule ordonnée, mais

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

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.

* *
*