Devoir surveillé n° 05 — 2h

LA CALCULATRICE N'EST PAS AUTORISÉE.

Les cinq parties sont entièrement indépendantes. On conseille cependant de les traiter dans l'ordre. On indiquera très précisément à quel endroit se situent les différentes parties dans la copie et on prendra soin de grouper toutes les réponses à une même partie dans la copie.

On attachera un réel soit à la présentation et à la lisibilité de la copie, sans quoi les questions ne seront pas même lues. N'oubliez pas de commenter et d'expliquer vos programmes si ce que réalisent ceux-ci n'est pas complètement évident.

Les consignes distribuées en début d'année s'appliqueront rigoureusement à ce devoir.

I Langages reconnaissables

Question 1

Énoncer précisément le *lemme de l'étoile*.

Question 2

On pose $\Sigma = \{a, b\}$. Déterminer parmi les langages suivants lesquels sont réguliers. Justifier.

1. $L_1 = \{a^n b^2 \mid n \in \mathbb{N}^*\}$
2. $L_2 = \{a^n b^{2n} \mid n \in \mathbb{N}^*\}$
3. $L_3 = \{uv \mid u, v \in \Sigma^*\}$
4. $L_4 = \{uu \mid u \in \Sigma^*\}$

II Déduction naturelle

Question 3

Démontrer que les séquents suivants sont dérivables.

1. En logique minimale.
$$A \wedge B, B \wedge C \vdash A \wedge C$$
2. En logique minimale. On pourra noter $\Gamma = \{A \to (B \vee C), \neg B, \neg C, A\}$.
$$A \to (B \vee C), \neg B, \neg C \vdash \neg A$$
3. En logique intuitionniste. On pourra noter $\Gamma = \{(A \vee C) \to (B \vee C), \neg C, A\}$.
$$(A \vee C) \to (B \vee C), \neg C \vdash A \to B$$
4. En logique classique. On pourra noter $\Gamma = \{C \to B, \neg C \to \neg A, A\}$.
$$C \to B, \neg C \to \neg A \vdash A \to B$$

III Chemin de largeur maximale

Soit $G = (S, A, p)$ un graphe pondéré non orienté connexe. On définit la *largeur* d'un chemin $c = s_0 s_1 \dots s_n$ comme la quantité

$$l(c) = \min_{0 \leq i \leq n-1} p(\{s_i, s_{i+1}\})$$

c'est-à-dire le poids minimal d'une arête de ce chemin. Un *chemin de largeur maximale* entre x et y est un chemin c qui relie x à y dont la largeur est maximale parmi tous les chemins entre x et y.

Par exemple, considérons le graphe de la figure 1, qui pourrait modéliser un réseau, le poids de chaque arête représentant la capacité de cette arête. La largeur d'un chemin est contrainte par l'arête de capacité minimale de ce chemin. Un chemin de largeur maximale est ainsi un chemin permettant de maximiser la bande passante entre deux sommets, ce qui n'est pas sans intérêt pour les algorithmes de routage.

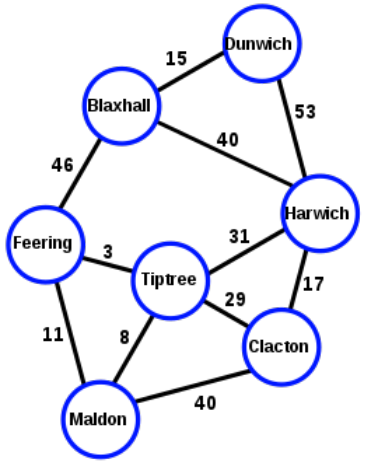


FIGURE 1 – Exemple de graphe modélisant un réseau. Le poids de chaque arête correspond à sa capacité.

Question 4

Déterminer un chemin de largeur maximale entre Maldon et Feering. Quelle est sa largeur ?

Question 5

Proposer un algorithme, que l'on décrira sommairement, permettant de trouver un arbre couvrant de poids maximal d'un graphe. Donner, sans justifier, sa complexité. Dérouler cet algorithme sur le graphe de la figure 1 pour en trouver un arbre couvrant de poids maximal.

Question 6

Soit $T = (S, B)$ avec $B \subseteq A$ un arbre couvrant de poids maximal de G. Montrer que pour tous sommets $x, y \in S$, T contient un des chemins de largeur maximale dans G entre x et y.

Question 7

En déduire un algorithme, que l'on décrira sommairement, qui permet de trouver un chemin de largeur maximale entre deux sommets x et y et donner sa complexité.

Cette dernière partie est indépendante des trois questions précédentes.

Question 8

On suppose que l'on connaît la largeur d'un chemin de largeur maximale entre x et y. Expliquer comment on peut alors construire un chemin de largeur maximale entre x et y en temps $O(|A|)$ linéaire en la taille[1] du graphe. *On n'utilisera donc pas l'algorithme précédent.*

Il existe un algorithme linéaire pour trouver un chemin de largeur maximale, du à Abraham PUNNEN *(1989), qui utilise cette dernière idée, mais ceci dépase le cadre de ce devoir.*

IV Problème 2-SAT (en CAML)

Dans cette partie on se propose de montrer que le problème 2-SAT peut être résolu par un algorithme en temps polynomial — et même linéaire en la taille de la formule.

Une formule booléenne est dite en k-FNC si elle est en forme normale conjonctive, c'est-à-dire une conjonction de disjonction de littéraux, et que chaque clause contient exactement k littéraux. Par exemple, les deux formules suivantes sont en 2-FNC.

- $\varphi_0 = (\neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_3) \wedge (v_3 \vee v_3) \wedge (v_2 \vee v_1) \wedge (\neg v_2 \vee v_3)$
- $\varphi_1 = (v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_2) \wedge (v_1 \vee \neg v_3)$

Le problème 2-SAT consiste à déterminer si une formule en 2-FNC est satisfiable. On notera n le nombre de variables qui apparaissent dans la formule et m son nombre de clauses. On supposera toujours que $n > 0$ et $m > 0$, une formule sans clauses étant toujours satisfiable. On suppose également qu'aucune clause ne contient un littéral et son opposé, sinon la formule n'est évidemment pas satisfiable. Enfin, on suppose que toutes les clauses sont deux à deux distinctes, en supposant que $(v_i \vee v_j)$ et $(v_j \vee v_i)$ représente la même clause.

On suppose par la suite, quitte à réindexer, que l'ensemble des variables qui apparaîssent dans une formule est de la forme $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, avec $n \geq 1$.

1. Le graphe est connexe donc $|S| = O(|A|)$.

On représente les formules sous forme 2-FNC en CAML en utilisant les types suivants :

OCAML
```
type litteral = Pos of int | Neg of int
type clause = litteral * litteral
type formule = clause list
```

Par exemple, la formule φ_0 est représentée par l'expression suivante :

OCAML
```
let phi0 = [(Neg 2, Neg 3); (Neg 1, Pos 3); (Pos 3, Pos 3);
            (Pos 2, Pos 1); (Neg 2, Pos 3)]
```

On rappelle que l'on note $\overline{\ell}$ la négation d'un littéral ℓ, c'est-à-dire $\overline{\ell} = \neg v$ si $\ell = v$ et $\overline{\ell} = v$ si $\ell = \neg v$.

Question 9

Écrire une fonction `label : litteral -> int` qui renvoie l'indice de variable d'un littéral et une fonction `neg : litteral -> litteral` qui renvoie le littéral opposé d'un littéral donné.

Question 10

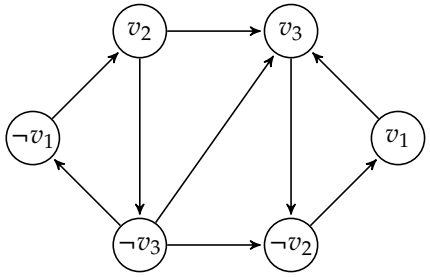
Écrire une fonction `nb_variables : formule -> int` qui renvoie le nombre n de variables qui apparaissent dans une 2-FNC. On rappelle que l'on suppose que la formule comporte exactement les variables $v_1, v_2, \dots, v_n$ et il suffit donc simplement de renvoyer l'indice maximal trouvé.

Remarquons qu'une clause $(\ell_1 \vee \ell_2)$ peut se mettre sous une forme implicative $\overline{\ell_1} \to \ell_2$ ou $\overline{\ell_2} \to \ell_1$, qui lui sont sémantiquement équivalentes. Nous allons construire un graphe dont les sommets sont les littéraux sur les variables de φ et traduisant toutes ces implications sur les littéraux.

Soit φ une formule en 2-FNC à m clauses sur l'ensemble de variables $\mathcal{V} = \{v_1, \dots, v_n\}$. On définit le **graphe d'implication** G_φ de φ comme le graphe *orienté* $G_\varphi = (S, A)$ avec :

$$S = \mathcal{V} \cup \{\neg v \mid v \in \mathcal{V}\} \qquad A = \bigcup_{(\ell_1 \vee \ell_2) \text{ clause de } \varphi} \left\{(\overline{\ell_1}, \ell_2), (\overline{\ell_2}, \ell_1)\right\}$$

Par exemple, le graphe d'implication G_{φ_0} est représenté ci-dessous :



Question 11

Représenter le graphe G_{φ_1}.

Question 12

Montrer que la taille du graphe d'implication G_φ est linéaire en la taille de φ. Préciser le nombre de sommets et le nombre d'arcs.

On représente un graphe par liste d'adjacences en utilisant le type suivant :

OCAML
```
type graphe = int list array
```

Les variables propositionnelles étant $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ on se donne la convention suivante : on représente le littéral v_i par le sommet $2i - 2$ et le littéral $\neg v_i$ par le sommet $2i - 1$.

Question 13

Écrire une fonction `graphe_implication : formule -> graphe` qui créé le graphe d'implication d'une formule passée en paramètre. Cette fonction doit avoir une complexité linéaire en la taille de la formule. On rappelle que l'on suppose que toutes les clauses sont distinctes.

Question 14

Soit $(\ell_1, \ell_2) \in A$ un arc du graphe d'implication. Montrer que $\varphi \vDash \ell_1 \to \ell_2$.

Question 15

Soit $\ell_1, \ell_2 \in S$ deux sommets du graphe. Montrer que s'il existe un chemin de ℓ_1 à ℓ_2 dans G_φ, alors $\varphi \vDash \ell_1 \to \ell_2$.

Question 16

En déduire que si φ est satisfiable, aucune composante fortement connexe de G_φ ne contient à la fois un littéral et sa négation.

On va montrer que la réciproque est également vraie et en déduire une construction d'une valuation qui satisfait φ.

Soit $\mu : \mathcal{V} \to \{0,1\}$ une valuation. On note encore μ le prolongement de cette valuation aux formules.

Question 17

Montrer que si φ est satisfiable, alors μ est constante sur chaque composante fortement connexe de G_φ, c'est-à-dire que tous les littéraux d'une même composante fortement connexe ont la même valeur de vérité.

On suppose désormais que dans le graphe G_φ aucune composante fortement connexe ne contient à la fois un littéral et sa négation.

Remarquons que le graphe est antisymétrique : si $(\ell_1, \ell_2) \in A$ alors $(\overline{\ell_2}, \overline{\ell_1}) \in A$. Ainsi, si $\ell_1, \ell_2 \in S$ sont deux sommets de G_φ, il existe un chemin de ℓ_1 à ℓ_2 si et seulement s'il en existe un de $\overline{\ell_2}$ à $\overline{\ell_1}$. Il s'ensuit que l'ensemble des négations des littéraux d'une composante fortement connexe C est encore une composante fortement connexe que l'on note $\overline{C}$.

Remarquons également s'il existe un chemin d'un composante fortement connexe C à une autre C', ce que l'on note $C \to C'$ alors on ne peut pas affecter la valeur vraie à C et fausse à C'. Ainsi, si C est une composante fortement connexe qui est un *puits* du méta-graphe des composantes fortement connexes, on a tout intérêt à affecter la valeur de vérité 1 à tous ses littéraux. Ceci impose que $\overline{C}$, qui est une composante fortement connexe *source*, verra tous ses littéraux évalués à 0, ce qui ne pose pas de problème pour les implications.

On propose alors l'algorithme suivant pour construire une valuation μ qui satisfait φ. On commence par une valuation μ qui n'affecte encore aucune valeur de vérité à aucune variable.

- Tant qu'il reste des sommets :
  - choisir C une composante fortement connexe puits (on a donc $\overline{C}$ qui est une source) ;
  - affecter une valeur de vérité à toutes les variables propositionnelles de C de telle manière à avoir $\mu(\ell) = 1$ pour tout sommet ℓ de C ; on aura donc $\mu(\overline{\ell}) = 0$ pour tout sommet ℓ de C et donc tous les littéraux de $\overline{C}$ auront pour valeur de vérité 0 ;
  - supprimer tous les sommets de C et $\overline{C}$.

Par exemple, pour le graphe G_{φ_0}, cet algorithme commence par la composante fortement connexe $\{v_1, \neg v_2, v_3\}$ et affecte donc $\mu(v_1) = \mu(v_3) = 1$ et $\mu(v_2) = 0$ puis supprime les sommets de la composante fortement connexe $\{v_1, \neg v_2, v_3\}$ et de sa composantes opposée, c'est-à-dire en fait ici tous les sommets. On vérifie que l'on a bien $\mu \vDash \varphi$.

Question 18

Quel algorithme peut-on utiliser pour trouver les composantes fortements connexes ? Décrire rapidement mais précisément le principe de cet algorithme. Quelle est sa complexité ?

Question 19

Comment implémenter efficacement la recherche d'une composante fortement connexe puits ? On rappelle que l'on souhaite une complexité linéaire pour la méthode complète.

Question 20

Montrer que cet algorithme se termine et que la valuation μ renvoyée est bien définie pour toute variable de $\mathcal{V}$.

Question 21

Montrons que μ est bien un modèle de φ, c'est-à-dire que chaque clause est bien satisfaite.

On a donc montré que la formule φ est satisfiable si et seulement si aucune composante fortement connexe de G_φ ne contient à la fois un littéral et sa négation. On a également montrer, dans le cas où φ est satisfiable, comment en construire un modèle en complexité $O(n+m)$ linéaire en la taille de la formule.

On supposer disposer d'une fonction `composantes_fortement_connexes : graphe -> int array` qui renvoie un tableau associant à chaque sommet l'identifiant unique de sa composante fortement connexe. On suppose que la complexité de cette fonction est en $O(|S| + |A|)$.

Question 22

Écrire une fonction `satisfiable : formule -> bool` qui indique si une formule est satisfiable ou non. On demande une complexité linéaire en la taille de la formule $O(n + m)$ où n est le nommbre de variables et m le nombre de clauses. On justifiera rapidement.

V Caractérisation des programmes bien typés

Le langage CAML utilise un système d'inférence de type pour vérifier que les expressions du langage sont bien typées. Il est possible de formaliser le caractère bien typé des expressions à l'aide d'un système de règles. Dans cette partie, on se propose de manipuler un tel système pour un petit fragment du langage CAML, ne contenant que les entiers, les booléens et les fonctions.

Nous allons établir une relation entre les expressions bien typées $\mathcal{E}$ et les types $\mathcal{T}$. Pour une expression bien typée $\texttt{e} \in \mathcal{E}$ et un type $\tau \in \mathcal{T}$ nous noterons $\vdash \texttt{e} : \tau$ le jugement correspondant au fait que l'expression `e` est de type τ. Ce type de relation sera appelé un *jugement* de type. Par exemple, on aura le jugement $\vdash$ `42` : *int* et le jugement $\vdash$ `true` : *bool*.

Par ailleurs, partant de deux expressions `f` et `e` pour lesquelles on aurait justifié $\vdash \texttt{f} : \sigma \to \tau$ et $\vdash \texttt{e} : \sigma$, alors on en *déduit* que l'application de `f` à `e` est légitime et on obtient le jugement $\vdash$ `(f e)` : τ. Par exemple, à partir du jugement $\vdash$ `succ` : *int* → *int* et du jugement $\vdash$ `42` : *int* on en déduit le jugement $\vdash$ `(succ 42)` : *int*. Cette combinaison sera à l'origine d'une *règle*.

1. Donner, sans justifier, le *jugement* que l'on obtiendrait pour l'*expression* `(max 42)`. *On ne demande pas d'évaluer cette expression.*

Il nous reste à déterminer comment doivent être typées les variables et garantir qu'une même variable est toujours typée de la même manière. Par exemple pour émettre un jugement de type pour l'expression `fun x -> e`, dont le type est de la forme $\sigma \to \tau$, avec `e = (x = x)` dont le type doit donc être τ, il faut garantir que les occurrences de x dans e doivent être de type σ. On introduit pour cela un *environnement de typage* ou *contexte*, noté Γ, qui associe à certains noms de variable un type. Le *jugement* de typage sera donc de la forme

$$\Gamma \vdash \texttt{e} : \tau$$

pour sigifier : « si chaque variable x de e a le type indiqué par l'environnement Γ, alors l'expression e est bien typée et a le type τ. Ici, on aurait par exemple $\Gamma \vdash$ `e` : *bool*, avec $\Gamma = \{\texttt{x} : \tau\}$.

On introduit maintenant les règles appelées *règles de typage*.

Constantes : Les axiomes indiquent que les *constantes* du langage ont bien le type attendu. Il y en a une infinité, puisqu'il y a une règle pour chaque entier.

$$\frac{}{\Gamma \vdash \texttt{true} : \mathit{bool}}\ \text{cste} \qquad \frac{}{\Gamma \vdash \texttt{false} : \mathit{bool}}\ \text{cste} \qquad \frac{}{\Gamma \vdash n : \mathit{bool}}\ \text{cste} \quad \text{pour tout } n \in \mathbb{N}$$

Variables : Un autre axiome fondamental permet d'associer son type à une variable en consultant simplement l'environnement. Comme en déduction naturelle $\Gamma, \texttt{x} : \tau$ désigne $\Gamma \cup \{\texttt{x} : \tau\}$.

$$\frac{}{\Gamma, \texttt{x} : \tau \vdash \texttt{x} : \tau}\ \text{var}$$

Fonctions : Enfin, nous ajoutons deux règles relatives à l'abstraction permettant de définir les fonctions et à leur application à des arguments.

$$\frac{\Gamma, \texttt{x} : \sigma \vdash \texttt{e} : \tau}{\Gamma \vdash \texttt{fun x -> e} : e : \sigma \to \tau}\ \text{abstr} \qquad \frac{\Gamma \vdash \texttt{f} : \sigma \to \tau \quad \Gamma \vdash \texttt{e} : \sigma}{\Gamma \vdash \texttt{(f e)} : \tau}\ \text{appli}$$

On considère l'environnement de typage $\Gamma = \{\texttt{f} : \alpha \to (\beta \to \gamma), \texttt{g} : \beta \to \alpha\}$ où α, β et γ sont des types arbitraires. La dérivation suivante permet d'attester que l'expression `fun x -> f (g x) x` est bien typée de type $\beta \to \gamma$ dans l'environnement de typage Γ.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \texttt{x} : \beta \vdash \texttt{f} : \alpha \to (\beta \to \gamma)}\ (4) \qquad \dfrac{\dfrac{}{\Gamma, \texttt{x} : \beta \vdash \texttt{x} : \beta}\ (7) \qquad \dfrac{}{\Gamma, \texttt{x} : \beta \vdash \texttt{g} : \beta \to \alpha}\ (6)}{\Gamma, \texttt{x} : \beta \vdash \texttt{g x} : \alpha}\ (5)}{\Gamma, \texttt{x} : \beta \vdash \texttt{f (g x)} : \beta \to \gamma}\ (3) \qquad \dfrac{}{\Gamma, \texttt{x} : \beta \vdash \texttt{x} : \beta}\ (8)}{\Gamma, \texttt{x} : \beta \vdash \texttt{f (g x) x} : \gamma}\ (2)}{\Gamma \vdash \texttt{f (g x) x} : \beta \to \gamma}\ (1)$$

Question 23

Le nom des règles utilisées n'a pas été indiqué sur l'arbre ci-dessus. Indiquer, pour chaque règle entre (1) et (8) le nom de la règle utilisée. Il est inutile de recopier l'arbre en entier, il suffit d'indiquer en face de chaque indice (i) le nom de la règle.

On indiquera impérativement par la suite le nom des règles utilisées.

Question 24

Montrer que les jugements de type suivants sont dérivables :

1. On pourra noter $\Gamma = \{\texttt{f} : \alpha \to \beta, \texttt{x} : \alpha\}$

$$\vdash \texttt{fun f -> fun x -> f x} : (\alpha \to \beta) \to \alpha \to \beta$$

2. On note $\Gamma = \{\texttt{succ} : int \to int\}$ et on pourra noter $\Delta = \Gamma \cup \{\texttt{x} : int\}$.

$$\Gamma \vdash \texttt{(fun f -> succ x) 42} : int$$

Inversement, on peut également constater que ces règles ne permettent pas de dériver un jugement de typage pour des expressions incohérentes.

Considérons l'expression `e = (fun f -> f 1 2) (fun x -> 3)`. Nous allons raisonner par l'absurde pour montrer que cette expression n'est pas typable par un jugement de typage. Supposons donc par l'absurde que l'on a réussi à dériver le jugement de typage $\vdash e : \alpha$ pour un certain type α. On pourra remarquer par la suite le fait que le type int ne peut pas être décomposé comme un type de la forme $\beta \to \gamma$.

Question 25

Quelle est la seule règle qui a pu être utilisée en dernier pour obtenir ce jugement ?

On remarque alors qu'il est nécessaire de fournir des dérivations pour les jugements : $\vdash$ `fun x -> 3` $: \beta$ et $\vdash$ `fun f -> f 1 2` $: \beta \to \alpha$ pour un certain type β.

Question 26

Considérons d'abord le jugement $\vdash$ `fun x -> 3` $: \beta$. De quelle forme doit alors être le type β ?

Question 27

En procédant par conditions nécessaires sur le schéma de dérivation qui permettrait d'obtenir $\vdash$ `fun f -> f 1 2` $: \beta \to \alpha$, montrer que nécessairement, $\beta = int \to (int \to \alpha)$.

Question 28

Conclure.