Devoir surveillé n°6

Des formes normales et des arbres de mots

Consignes

La durée du devoir est de 4 h. Les documents, notes de cours, calculatrices et tous dispositifs électroniques sont interdits et doivent être éteint et rangés dans vos sacs. Ce sujet comprend 6 pages. **Vos réponses doivent être rigoureusement écrites et justifiées, même si la question ne le précise pas.** Votre nom doit être noté sur toutes vos copies, chaque page doit être numérotée sous la forme k/n avec n le nombre total de pages utilisées. L'ensemble de vos copies doit être glissé, sans le sujet, dans votre première copie double. Vos résultats doivent être soulignés ou encadrés, vous pouvez également souligner ou encadrer des résultats intermédiaires si vous voulez les mettre en valeur. Un espace de quelques lignes doit être laissé vide au début de la première page, une marge à gauche doit être réservée sur chacune des pages. Le non-respect de ces consignes pourra être l'objet d'un malus de points.
Les questions de programmation sont toutes à traiter en OCaml ou en C : toutes les fonctions des sections A.1 et B.1 "Traits et éléments techniques à connaître" du programme de la MP2I/MPI sont autorisées sans rappel. Les fonctions des sections A.2 et B.2 "Éléments techniques devant être reconnus et utilisables après rappel" du programme de la MP2I/MPI le sont si leurs types ou prototype, si besoin leurs complexités, sont rappelées. Les autres ne sont pas autorisées (sauf si l'énoncé les fournit). Pour C, les bibliothèques `<assert.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdio.h>`, `<stdlib.h>` sont supposées chargées. Le langage attendu à une question est précisé par l'utilisation de la syntaxe OCaml ou C pour définir le type ou prototype de fonction attendu.

Bon courage !

Le sujet est divisé en deux problèmes entièrement indépendants. Le premier problème porte sur la manipulation de formes normales conjonctives en C, la seconde partie propose l'implémentation d'ensemble de mots binaires en OCaml par des arbres de mots.

I Autour de la forme normale conjonctive (C)

I.A Représentation des formules sous forme normale conjonctive

On note $\mathcal{P}$ un ensemble de variables propositionnelles. Un **littéral** est soit une variable propositionnelle (dit littéral **positif**), soit la négation d'une variable propositionnelle (dit littéral **négatif**). Pour l un littéral, on note $|l|$ la variable propositionnelle sous-jacente. Une **occurrence** d'un littéral l dans une formule est une occurrence de la variable propositionnelle $|l|$ dans la formule.

On rappelle qu'une **clause** est une disjonction de littéraux et une formule sous **forme normale conjonctive** est une conjonction de clauses disjonctives. Une clause est dite **unitaire** si elle ne contient qu'un seul littéral.

Nous allons représenter une formule sous forme normale conjonctive par la structure C suivante :

C
```c
#define MAX_VP 100
typedef int litteral;
typedef struct {
    litteral* litteraux;
    int nb_litteraux; } clause;
typedef struct {
    clause* clauses;
    int nb_clauses; } cnf;
```

La première instruction définit la constante littérale entière `MAX_VP`. La structure `clause` représente une clause, avec un tableau de littéraux `litteraux` de longueur `nb_litteraux`. Chaque littéral est un entier k tel que $|k| \in [\![1, MAX\_VP]\!]$, avec $k > 0$ représentant le littéral positif p_k et $k < 0$ représentant le littéral négatif $\neg p_k$. La structure `cnf` représente une formule sous forme normale conjonctive, avec un tableau de clauses `clauses` de longueur `nb_clauses`.

La fonction `int abs(int x)` de la bibliothèque `stdlib.h` renvoie la valeur absolue de l'entier en entrée. Dans le reste du problème, on ignore les problématiques de libération de la mémoire allouée sur le tas en C.

▶ **Question 1** Écrire une fonction `int taille(cnf f)` qui prend en entrée une formule `f` et qui renvoie le nombre de littéraux de la formule (en comptant les répétitions).

▶ **Question 2** Écrire une fonction `int nb_clauses_unitaires(cnf f)` qui prend en entrée une formule `f` et qui renvoie le nombre de clauses unitaires de la formule.

▶ **Question 3** Écrire une fonction `int nb_occurrences(cnf f, litteral l)` qui prend en entrée une formule `f` et un littéral `l` et qui renvoie le nombre d'occurrences de `l` dans la formule.

On définit une valuation en C par un tableau de booléens `bool* v` de taille `MAX_VP + 1`, la première case contenant une valeur quelconque et `v[i]` contenant la valeur de vérité de p_i pour $i \in [\![1, MAX\_VP]\!]$.

▶ **Question 4** Écrire une fonction `bool satisfait(cnf f, bool* v)` qui prend en entrée une formule `f` et une valuation `v` et qui renvoie `true` si la valuation `v` satisfait la formule `f`, `false` sinon.

> *Solution* Correction dans le fichier source C.
> On le justifie comme suis : une valuation ν satisfait une clause c si et seulement si il existe un littéral l de c tel que $\nu \vDash l$. Ainsi, une valuation ν satisfait une formule f si et seulement si elle satisfait toutes les clauses de f. Ainsi, la fonction `bool satisfait(cnf f, bool* v)` renvoie `true` si et seulement si pour toute clause c de f, il existe un littéral l de c tel que $\nu \vDash l$.
> La fonction `bool satisfait_clause(clause c, bool* v)` renvoie `true` si et seulement si $\nu \vDash c$.

▶ **Question 5** Déterminer précisément sa complexité.

> *Solution* La fonction `bool satisfait(cnf f, bool* v)` parcourt chaque clause de la formule `f` et pour chaque littéral de chaque clause, elle effectue un nombre constant d'opération (accès tableau). Ainsi, la boucle interne effectue $\lambda|c[i]|$ opérations et la boucle externe effectue $\sum_{i=0}^{nb\_clauses-1} \lambda|c[i]| = \lambda|f|$. Ainsi, la complexité de la fonction `bool satisfait(cnf f, bool* v)` est en $\mathcal{O}(|f|)$.

▶ **Question 6** En considérant qu'itérer sur toutes les valuations possibles prend un temps constant par valuation, déterminer la complexité d'un algorithme de recherche de valuation satisfaisant une formule sous forme normale conjonctive par force brute. *On exprimera cette complexité en fonction du nombre de variables propositionnelles considérées et de la taille de la formule.*

> *Solution* Pour une formule f de taille $|f|$ et n variables propositionnelles, il y a 2^n valuations possibles. Ainsi, la complexité d'un algorithme de recherche de valuation satisfaisant une formule sous forme normale conjonctive par force brute est en $\mathcal{O}(|f|2^n)$.

I.B Validité d'une formule sous forme normale conjonctive

▶ **Question 7** Soit c une clause. Déterminer et justifier une condition nécessaire et suffisante pour que c soit valide.

> *Solution* Montrons que c est valide si et seulement s'il existe $p \in \mathcal{P}$ tel que c contient à la fois une occurrence positive et négative de p.
> ⇒
> Montrons-le par contraposée. Soit c une clause telle que pour toute variable propositionnelle $p \in \mathcal{P}$, c ne contient pas à la fois une occurrence positive et négative de p. Posons la valuation suivante :
>
> $$\nu : p \in \mathcal{P} \mapsto \begin{cases} \texttt{faux} & \text{si } c \text{ contient une occurrence positive de } p; \\ \texttt{vrai} & \text{sinon } (c \text{ contient une occurrence négative de } p \text{ ou n'en contient pas}). \end{cases}$$
>
> Soit l un littéral de c. Si l est un littéral positif de la forme $p \in \mathcal{P}$, alors $\nu(p) = \texttt{faux}$ donc $\nu \nvDash l$. Si l est un littéral négatif de la forme $\neg p$, alors $\nu(p) = \texttt{vrai}$ donc $\nu \nvDash l$. La valuation ν ne satisfait aucun littéral de la clause c, donc c n'est pas valide.
> ⇐
> Soit $p \in \mathcal{P}$ tel que c contient à la fois une occurrence positive et négative de p. Alors pour toute valuation

$\nu \in \mathbb{B}^{\mathcal{P}}$, si $\nu(p) =$ `vrai`, alors $\nu \vDash p$ donc $\nu \vDash c$ et si $\nu(p) =$ `faux`, alors $\nu \vDash \neg p$ donc $\nu \vDash c$. Donc pour toute valuation $\nu \in \mathbb{B}^{\mathcal{P}}$, $\nu \vDash c$. Donc c est valide.

▶ **Question 8** Soient φ, ψ deux formules quelconques. Montrer que $\varphi \wedge \psi$ est valide si et seulement si φ et ψ sont valides.

Solution Notons $Mod(\varphi)$ l'ensemble des modèles de φ. Alors on a :

$$\begin{aligned} \vDash \varphi \wedge \psi \text{ ssi } & Mod(\varphi \wedge \psi) = \mathcal{V} \\ \text{ssi } & Mod(\varphi) \cap Mod(\psi) = \mathcal{V} \\ \text{ssi } & Mod(\varphi) = \mathcal{V} \text{ et } Mod(\psi) = \mathcal{V} && \text{car } Mod(\varphi), Mod(\psi) \subseteq \mathcal{V} \\ \text{ssi } & \vDash \varphi \text{ et } \vDash \psi. \end{aligned}$$

▶ **Question 9** En déduire en le justifiant une condition nécessaire et suffisante pour qu'une formule sous forme normale conjonctive f soit valide.

Solution Soit $f = \bigwedge_{i=1}^{k} c_i$ une formule sous forme normale conjonctive avec $c_1 \dots c_k$ ses clauses. En généralisant la Question 8 à une conjonction arbitrairement grande de formules, on sait que f est valide si et seulement si pour tout $i \in [\![1, k]\!]$, c_i est valide. Ainsi, par la Question 7, on a que f est valide si et seulement si pour tout $i \in [\![1, k]\!]$, il existe $p \in \mathcal{P}$ tel que c_i contient à la fois une occurrence positive et négative de p.

▶ **Question 10** Écrire une fonction `bool est_valide(cnf f)` qui prend en entrée une formule `f` et qui renvoie `true` si la formule est valide, `false` sinon. Préciser sans la justifier sa complexité temporelle et spatiale.

Solution La complexité temporelle est en $\mathcal{O}\big(MAX\_VP \times |f|\big)$ et la complexité spatiale est en $\mathcal{O}(MAX\_VP)$.

I.C Clauses d'Horn et satisfaisabilité

Définition des clauses d'Horn Dans cette section, on se restreint à manipuler des formules sous forme normale conjonctive ne contenant que des clauses d'Horn :

Définition 1 – Clause et formule d'Horn

Une **clause d'Horn** est une clause qui contient au plus un littéral positif. Une clause d'Horn est dite **négative pure** si elle ne contient que des littéraux négatifs, **implication** si elle contient un littéral positif (et éventuellement aucun littéral négatif). Une formule d'Horn est une formule sous forme normale conjonctive dont toutes les clauses sont des clauses d'Horn.

▶ **Question 11** Pour les formules suivantes, déterminer celles qui sont des clauses d'Horn. On justifiera seulement les réponses négatives.

1. $x \vee y \vee z$;
2. $\neg x \vee z \vee \neg y$;
3. $x \to \neg y \vee \neg z$;
4. $x \wedge \neg y \vee z \vee \neg x$;
5. $\neg x \vee \neg y \vee \neg z$;
6. $\neg x \vee \neg(y \to z)$.

Solution *On justifie ici même si ce n'était pas demandé, au cas où vous ne comprendriez pas votre éventuelle erreur.*

1. Ce n'est pas une clause d'Horn : elle contient trois littéraux positifs ;
2. c'est une clause d'Horn, une clause avec un littéral positif et deux négatifs ;
3. ce n'est pas une clause d'Horn : elle contient deux littéraux positifs et une implication (donc ce n'est

même pas une clause) ;
4. ce n'est pas une clause d'Horn : elle contient une conjonction donc ce n'est même pas une clause ;
5. c'est une clause d'Horn, une clause avec aucun littéral positif ;
6. ce n'est pas une clause d'Horn : elle contient une implication donc ce n'est même pas une clause.

▶ **Question 12** Montrer que pour toute clause d'Horn, il existe une formule équivalente à cette clause sous la forme d'une implication entre :
— à gauche, une conjonction de variables propositionnelles ou $\top$;
— à droite, une variable propositionnelle ou $\bot$;
où $\top$ (resp. $\bot$) est une formule satisfaite par toute valuation (resp. insatisfaite par toute valuation).

Solution Soit c une clause d'Horn.
— Si cette clause est une implication unitaire, alors elle est de la forme $p \equiv \underline{\top \to p}$;
— si cette clause est une implication non unitaire, alors elle est de la forme (quitte à réordonner les littéraux) $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k$, qui est équivalente à $\underline{p_1 \wedge p_2 \wedge \ldots \wedge p_k \to p}$;
— si cette clause est négative pure (donc une conjonction de littéraux négatifs), alors elle est de la forme $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \equiv \underline{p_1 \wedge \ldots \wedge p_k \to \bot}$.

Application à un problème de décision Ces clauses d'Horn permettent d'énoncer des faits et des implications entre faits. On peut donc les utiliser pour modéliser des problèmes de décision concrets : on définit les variables propositionnelles $p_1, \ldots p_5$ représentant les énoncés suivants :
— p_1 : le meurtre a eu lieu dans la cuisine ;
— p_2 : la docteure était en mission hors de la maison à 20 h ;
— p_3 : le meurtre a eu lieu à 20 h ;
— p_4 : le colonel est innocent ;
— p_5 : la docteure est innocente.

▶ **Question 13** Exprimer sous la forme d'une clause d'Horn les énoncés suivants :
— si la docteure était hors de la maison à 20 h et si le meurtre a eu lieu à 20 h, alors la docteure est innocente ;
— la docteure était en mission hors de la maison à 20 h ;
— l'un des deux suspects est coupable ;
— le meurtre a eu lieu à 20 h.

Solution
— $\neg p_2 \vee \neg p_3 \vee p_5$;
— p_2 ;
— $\neg p_4 \vee \neg p_5$;
— p_3.

Satisfaisabilité de formules d'Horn par un algorithme glouton Il existe un algorithme glouton pour déterminer si une formule sous forme normale conjonctive dont les clauses sont d'Horn est satisfaisable, et qui permet même de construire une valuation la satisfaisant.

Pseudo-code

Requiert : f une conjonction de clauses d'Horn,
$\mathcal{P}$ l'ensemble des variables propositionnelles ayant une occurence dans f.

```
1 : Fonction HornSAT(f)
2 :     initialement, ν(p) = faux pour tout p ∈ P.
3 :     Tant que il existe c clause implication de f telle que ν ⊭ c , faire
4 :         p ← le littéral positif de c
5 :         ν(p) ← vrai
6 :     Pour chaque clause négative pure c de f , faire
7 :         Si ν ⊭ c alors
8 :             retourne la formule f n'est pas satisfaisable.
9 :     retourne ν
```

▶ **Question 14** Montrer la terminaison de cet algorithme.

Solution L'initialisation de ν termine, tout comme le corps de la boucle **Pour**. Ainsi, l'extérieur et le corps de la boucle **Tant que** termine.
Le nombre de variables propositionnelles fixées à faux par ν est un entier, minoré par 0 et diminue strictement à chaque tour de boucle. En effet, l'effet d'un tour de boucle est, pour une clause implication c telle que $\nu \not\models c$, de fixer ν à vraie pour la variable propositionnelle p apparaissant positivement dans c. Or, si $\nu \not\models c$, alors on a notamment $\nu \not\models p$ donc $\nu(p) = \texttt{faux}$ avant le tour de boucle et $\nu(p) = \texttt{vrai}$ après le tour de boucle. Ainsi, le nombre de variables propositionnelles fixées à faux par ν est un variant de boucle et le corps de la boucle termine. Ainsi, la boucle **Tant que** termine.
Ainsi, l'algorithme termine.

▶ **Question 15** Déterminer la complexité de cet algorithme en fonction de la taille de la formule en entrée.

Solution *Ici, la difficulté était d'évaluer une complexité avec peu d'informations sur l'implémentation des opérations décrites à l'intérieur.*
Déterminer le nombre de variables propositionnelles et initialiser la valuation en conséquence se fait en $\mathcal{O}(|f|)$, car ce nombre de variables propositionnelles n vérifie $n \leq |f|$.
La boucle **Tant que** effectue au plus n tours de boucle, car à chaque tour de boucle, on fixe une variable propositionnelle à vrai. Or, évaluer la condition de cette boucle consiste à parcourir chaque clause de f pour déterminer si elle est une clause implication et si la valuation actuelle ne la satisfait pas. Pour chaque clause c de f, on doit parcourir chaque littéral de c pour déterminer si c'est un littéral positif et s'il est satisfait par ν, ce qui a un coût en $\mathcal{O}(|c|)$. Ainsi, la condition de la boucle **Tant que** s'évalue en $\sum_{c \text{ clause de } f} \lambda|c| = \mathcal{O}(|f|)$, où λ est une constante. La boucle **Tant que** a donc une complexité en $\mathcal{O}(|f|^2)$.
Enfin, la boucle **Pour** parcours chaque clause de f et vérifie si elle est une clause négative pure et si elle est satisfaite par ν. Par le même raisonnement que précédemment, cela a un coût en $\mathcal{O}(|f|)$.
Ainsi, l'algorithme a une complexité en $\mathcal{O}(|f|^2)$.

▶ **Question 16** Que peut-on dire de la première clause d'Horn implication trouvé par l'algorithme HornSAT dans la boucle **Tant que** ?

Solution La première clause d'Horn implication trouvée par l'algorithme HornSAT dans la boucle **Tant que** est une clause implication c de f telle que $\nu \not\models c$. Or initialement on a $\nu(p) = \texttt{faux}$ pour tout $p \in \mathcal{P}$. Ainsi, la clause implication c ne contient pas de littéral négatif. Donc c est une clause unitaire.

▶ **Question 17** Montrer l'invariant suivant de la boucle **Tant que** :

« Pour tout $p \in \mathcal{P}$, si $\nu(p) = \texttt{vrai}$, alors pour toute valuation ν' modèle de f, $\nu'(p) = \texttt{vrai}$. »

Solution Initialement, pour tout $p \in \mathcal{P}$, $\nu(p) = \texttt{faux}$. Ainsi, l'invariant est vrai au début de la boucle.
Supposons que l'invariant est vrai au début d'un tour de boucle. Intéressons-nous à la clause implication c de f vérifiant $\nu \not\models c$, et posons $c = \neg p_1 \vee ... \vee \neg p_k \vee p$ (avec $k \in \mathbb{N}$). Alors ν ne satisfait aucun littéral de c, donc $\nu(p) = \texttt{faux}$ et pour tout $j \in [\![1, k]\!]$, $\nu(p_j) = \texttt{vrai}$. Ainsi, par hypothèse, toute valuation ν' modèle de f vérifie $\nu(p_j)$ pour tout $j \in [\![1, k]\!]$. ν' ne satisfait donc aucun littéral négatif de c : puisque $\nu' \models c$, on a donc $\nu'(p) = \texttt{vrai}$. Après le tour de boucle, l'invariant est donc toujours vrai.
Ainsi, la propriété est bien un invariant de boucle.

▶ **Question 18** En déduire la correction de l'algorithme HornSAT.

Solution Si HornSAT renvoie une valuation, alors c'est qu'il n'y a pas de clause implication c de f telle que $\nu \not\models c$ et qu'il n'y a pas de clause négative pure c de f telle que $\nu \not\models c$. Ainsi, ν satisfait toutes les clauses de f et donc $\nu \models f$.
Sinon, par l'invariant de boucle de la Question 17, à la fin de la boucle **Tant que** on a que pour toute valuation ν' modèle de f, pour tout $p \in \mathcal{P}$, si $\nu(p) = \texttt{vrai}$, alors $\nu'(p) = \texttt{vrai}$. Or, ici on a trouvé une clause négative pure c de f telle que $\nu \not\models c$: en notant $c = \neg p_1 \vee ... \vee \neg p_k$, on a $\nu(p_j) = \texttt{vrai}$ pour tout $j \in [\![1, k]\!]$. Ainsi, pour toute valuation ν' modèle de f, on a $\nu'(p_j) = \texttt{vrai}$ pour tout $j \in [\![1, k]\!]$, donc $\nu' \not\models c$ ce qui contredit $\nu' \models f$. Ainsi, f n'est pas satisfaisable.
Ainsi, l'algorithme HornSAT est correct.

▶ **Question 19** Utiliser l'algorithme HornSAT pour déterminer si la conjonction des clauses déterminées à la Question 13 est satisfaisable. En déduire le meurtrier.

Solution Notons $c_1 ... c_4$ les quatre clauses énoncées dans la Question 13. On applique l'algorithme HornSAT à la formule $f = c_1 \wedge c_2 \wedge c_3 \wedge c_4$.
On commence par $c_2 = p_2$, qui n'est pas satisfaite initialement. On pose donc $\nu_{p_2} = \texttt{vrai}$.
On continue avec $c_4 = p_3$, qui n'est pas satisfaite par ν. On pose donc $\nu_{p_3} = \texttt{vrai}$.
On continue avec $c_1 = \neg p_2 \vee \neg p_3 \vee p_5$, qui n'est pas satisfaite par ν. On pose donc $\nu_{p_5} = \texttt{vrai}$.
Les clauses implications sont maintenant toutes satisfaites par ν. Or, la seule clause négative pure est $c_3 = \neg p_4 \vee \neg p_5$, qui est satisfaite par ν. Ainsi, la formule est satisfaisable et le meurtrier est le colonel.

Implémentation de HornSAT ▶ **Question 20** Écrire une fonction `bool est_formule_horn(cnf f)` qui prend en entrée une formule `f` et qui renvoie `true` si `f` est une formule d'Horn, `false` sinon.

▶ **Question 21** Écrire une fonction `litteral get_litteral_positif(clause c)` qui prend en entrée une clause d'Horn `c` et qui renvoie `l` le littéral positif de `c` si elle est une clause implication, `0` si la clause est négative pure.

▶ **Question 22** Implémenter une fonction `bool* horn_sat(cnf f)` qui prend en entrée une formule `f` d'Horn et qui renvoie une valuation la satisfaisant si elle existe, `NULL` sinon.

Une représentation spécialisée des formules d'Horn par des graphes permet de résoudre le problème HornSAT en temps linéaire, ce problème est d'ailleurs P-complet. Cette excellente complexité permet d'appliquer l'algorithme à des problèmes de programmation logique. Le langage Prolog *utilise ces résultats en les étendant à la* logique du premier ordre*, ce qui permet une implémentation efficace d'algorithmes d'analyse syntaxique et lexicale, de résolution de requêtes déductives dans des bases de données, en intelligence artificielle, etc. Elles permettent aussi d'exprimer et d'analyser formellement de manière naturelle les systèmes de contrôle, les systèmes de production, ou encore les protocoles de communication comme avec logiciel* ProVerif.

II Facteurs de mots binaires (OCaml)

II.A Rappels, définitions et propriétés élémentaires

Langage OCaml

On rappelle que la fonction `List.rev : 'a list -> 'a list` renvoie la liste renversée de la liste passée en argument. De plus, la fonction `Array.of_list : 'a list -> 'a array` renvoie le tableau correspondant à la liste passée en argument. Enfin, la fonction `List.of_array : 'a array -> 'a list` renvoie la liste correspondant au tableau passé en argument.

Mots

Un **mot binaire** (dans la suite, simplement **mot**) est une suite finie $m = m_0 m_1 \dots m_{l-1}$ où chaque m_i est un symbole dans l'alphabet $\{0,1\}$ et ℓ est sa longueur. On note ε le mot vide, de longueur 0. On note $\ell(m)$ la longueur du mot m.

Pour deux mots $m = m_0 m_1 \dots m_{\ell-1}$ et $m' = m'_0 m'_1 \dots m'_{\ell'-1}$, on note : $m \cdot m' = m_0 m_1 \dots m_{\ell-1} m'_0 m'_1 \dots m'_{\ell'-1}$, la **concaténation** de m et m'. On note m^k le mot $m \cdot m \cdots \cdot m$ (k fois). On pourra omettre le point « $\cdot$ » dans la notation de la concaténation.

Pour $0 \leq i \leq j \leq \ell(m)$, on note $m[i : j]$ le mot $m_i m_{i+1} \dots m_{j-1}$, qui est appelé **facteur** de m. Un **préfixe** (resp. **suffixe**) de m est un facteur de la forme $m[0 : j]$ (resp. $m[i : \ell(m)]$). On remarque que ε est préfixe, suffixe et facteur de tout mot m (avec $i = j$, $j = 0$ ou $i = \ell(m)$). On note $m[: j] = m[0 : j]$ et $m[i :] = m[i : \ell(m)]$. Enfin, on note :

— $P(M) = \{m[: j] \mid m \in M, 0 \leq j \leq \ell(m)\}$ l'ensemble des préfixes de mots de M ;
— $S(M) = \{m[i :] \mid m \in M, 0 \leq i \leq \ell(m)\}$ l'ensemble des suffixes de mots de M ;
— $F(M) = \{m[i : j] \mid m \in M, 0 \leq i \leq j \leq \ell(m)\}$ l'ensemble des facteurs de mots de M.

En OCaml, les mots sont représentés par des tableaux ne contenant que les entiers 0 et 1. On utilisera le type OCaml suivant : `type mot = int array`.

Arbres binaires

On définit l'ensemble des arbres binaires par induction.

— L'arbre vide, noté V, est un arbre binaire ;
— si g et d sont deux arbres binaires et x une étiquette (dans ce sujet, les étiquettes sont des booléens), alors l'arbre binaire $N(x, g, d)$ est un arbre binaire.

On utilise ici le type d'arbre suivant :

```OCaml
type arbre = V | N of bool * arbre * arbre
```

▶ **Question 23** Implémenter deux fonctions `hauteur : arbre -> int` et `taille : arbre -> int` qui renvoient respectivement la hauteur et la taille (le nombre de nœuds) d'un arbre binaire. *On choisit ici la convention $h(V) = 0$.*

Dans la suite, on notera mathématiquement $t(a)$ la taille d'un arbre binaire a et $h(a)$ sa hauteur.

▶ **Question 24** Montrer que pour tout arbre binaire a, on a $t(a) \leq 2^{h(a)} - 1$.

Solution Montrons-le par induction sur a.

Initialisation : On a bien $t(V) = 0 \leq 0 = 2^0 - 1 = 2^{h(V)} - 1$.
Induction : Supposons que pour g, d deux arbres binaires la propriété est vérifiée. Soit x une étiquette. Alors on a :

$$\begin{aligned} t(N(x,g,d)) &= t(g) + t(d) + 1 \\ &\leq 1 + 2^{h(g)} - 1 + 2^{h(d)} - 1 + 1 && \text{par hypothèse d'induction} \\ &\leq 2^{\max(h(g),h(d))} + 2^{\max(h(g),h(d))} - 1 \\ &\leq 2^{1+\max(h(g),h(d))} - 1 = 2^{h(N(x,g,d))} - 1 \end{aligned}$$

Ainsi, la propriété est vérifiée pour $N(x,g,d)$.
Conclusion : Pour tout arbre binaire a, on a bien $t(a) \leq 2^{h(a)} - 1$.

▶ **Question 25** Sans le justifier, donner une valeur minimale de la taille d'un arbre binaire en fonction de sa hauteur.

Solution Pour tout arbre binaire a, on a $t(a) \geq h(A)$.

II.B Représentation d'un ensemble de mots par un arbre binaire

Dans cette partie, on utilise les arbres binaires tels que définis précédemment pour représenter un ensemble de mots. Un **arbre de mots** est un arbre binaire dont chaque nœud est étiqueté par un booléen. L'arbre de mots a représente l'ensemble de mots $\mathbb{M}(a)$ comme suit.

— Si $a = V$, alors $\mathbb{M}(a) = \varnothing$;
— si $a = N(b,g,d)$, alors $\mathbb{M}(a)$ est l'ensemble des mots composés de ε si b est vrai, les mots de la forme $0m$ pour tout $m \in \mathbb{M}(g)$ et les mots de la forme $1m$ pour tout $m \in \mathbb{M}(d)$.

Autrement dit, $\mathbb{M}(a)$ est l'ensemble des mots correspondant aux chemins de la racine de a aux nœuds étiquetés par `true`, en utilisant le caractère 0 quand on va à gauche et 1 quand on va à droite. On dessine graphiquement des arbres de mot comme suis, en écrivant T dans les nœuds étiquetés par `true` et F dans les nœuds étiquetés par `false`.

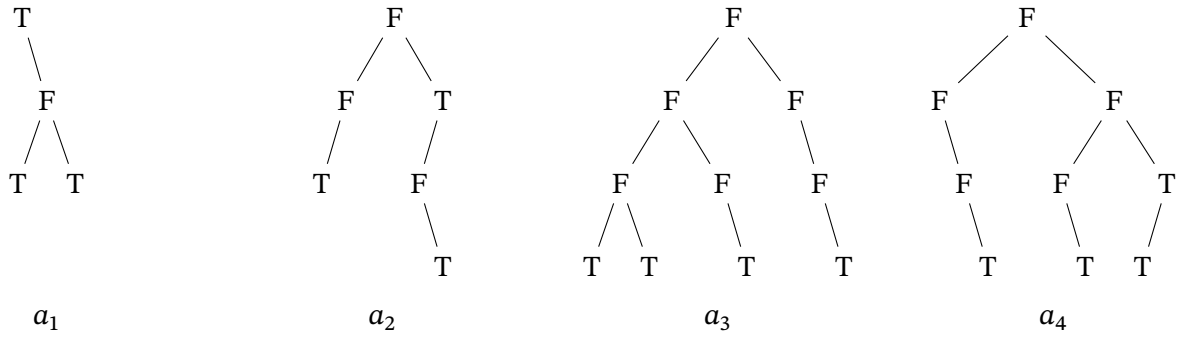


FIGURE 1 – Des arbres de mots

Ici, a_1 est l'arbre $N(T,V,N(F,N(T,V,V),N(T,V,V)))$. On a trois T dans l'arbre a_1, $\mathbb{M}(a_1)$ contient donc trois mots : dans l'ordre préfixe, ces mots sont ε, 10 et 11.

▶ **Question 26** Donner sans justifier l'ensemble de mots représenté par l'arbre a_2 et sa représentation par le type `arbre` en OCaml.

Solution a_2 représente l'ensemble de mots $\{00, 1, 101\}$. En OCaml, a_2 est représenté par :

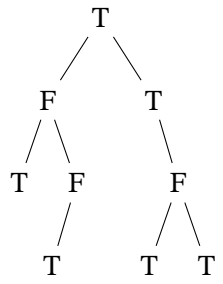
```ocaml
let a2 =
  N
    ( false,
      N (false, N (true, V, V), V),
      N (true, N (false, V, N (true, V, V)), V) )
```

On remarque qu'une infinité d'arbres de mots peuvent représenter un même ensemble de mots. En effet, il suffit de remplacer un arbre vide dans un arbre de mots par $N(\mathrm{F}, V, V)$ pour obtenir un arbre de mots représentant le même ensemble de mots. On appelle **arbre de mot réduit** un arbre de mot ne contenant aucune feuille $N(\mathrm{F}, V, V)$.

▶ **Question 27** Dessiner sans justifier un arbre de mots réduit représentant l'ensemble de mots suivant :

$$M = \{\varepsilon, 00, 010, 1, 110, 111\}.$$

Solution



▶ **Question 28** Montrer que tout arbre de mots a contenant un nœud étiqueté T vérifie $\mathbb{M}(a) \neq \emptyset$.

Solution Montrons-le par induction sur les arbres de mots.

Initialisation : L'arbre vide ne contient pas de nœud. La propriété est donc trivialement vraie.

Induction : Soit g, d deux arbres vérifiant la propriété (s'ils contiennent un nœud étiqueté T, alors ils représentent un ensemble de mots non vide). Soit x une étiquette et supposons que $N(x, g, d)$ contient une étiquette T. Alors :

— si x est vrai, alors $\varepsilon \in \mathbb{M}(N(x, g, d))$ donc $\mathbb{M}(N(x, g, d)) \neq \emptyset$;

— si x est faux, alors g ou d contient une étiquette T donc $\mathbb{M}(g) \neq \emptyset$ ou $\mathbb{M}(d) \neq \emptyset$. Ainsi, il existe $m \in \mathbb{M}(g)$ ou $m \in \mathbb{M}(d)$ tel que $0m \in \mathbb{M}(N(x, g, d))$ ou $1m \in \mathbb{M}(N(x, g, d))$ donc $\mathbb{M}(N(x, g, d)) \neq \emptyset$.

Conclusion : Pour tout arbre de mots a contenant un nœud étiqueté T, on a $\mathbb{M}(a) \neq \emptyset$.

▶ **Question 29** Montrer que, pour tout ensemble fini de mots M, il existe un unique arbre de mots réduit représentant M. *Cette question est difficile : on pourra montrer séparément l'existence et l'unicité, et il ne faut pas hésiter à proposer une stratégie de preuve même si l'on n'arrive pas à la mener à terme.*

Solution On distingue d'abord le cas particulier de l'ensemble vide $M = \emptyset$. Cet ensemble est représenté par l'arbre vide. Or, un arbre de mots réduit non vide contient au moins une feuille $N(\mathrm{T}, V, V)$ (puisque tout arbre binaire contient une feuille et que l'arbre de mots est réduit) donc ne peut pas représenter l'ensemble vide. Ainsi, l'unique arbre de mots réduit représentant l'ensemble vide est l'arbre vide.
Soit M un ensemble fini de mots, notons $\ell(M) = \max_{m \in M} \ell(m)$ la longueur maximale des mots de M, qui existe puisque M est fini. Montrons par récurrence forte sur $n \in \mathbb{N}$ la propriété suivante :

> « Pour tout ensemble de mots non vide M tel que $\ell(M) = n$, il existe un unique arbre de mots réduit représentant M. »

Initialisation : Pour $n = 0$, on a $\ell(M) = 0$ donc $M = \{\varepsilon\}$. L'arbre $N(\mathrm{T}, V, V)$ est bien et réduit et représente M : montrons qu'il est unique.
Pour tout arbre de mots réduit a représentant M, a est non vide (puisque $M \neq \emptyset = \mathbb{M}(V)$) donc est de la forme $N(x, g, d)$. Or, si $x = F$ alors on a $\mathbb{M}(a) = \{0m \mid m \in \mathbb{M}(g)\} \cup \{1m \mid m \in \mathbb{M}(d)\}$ qui ne peut donc pas contenir ε : par l'absurde, on a $x = T$. De plus, tous les mots de la forme $0m$ ou $1m$ sont différents de ε donc $\mathbb{M}(g) = \mathbb{M}(d) = \emptyset$, donc $g = d = V$ par la remarque précédente. Ainsi,

$a = N(\mathrm{T}, V, V)$ est bien l'unique arbre de mots réduit représentant M.

Hérédité : Soit $n \in \mathbb{N}^*$. Supposons que la propriété est vraie jusqu'au rang n exclu.

Soit M un ensemble de mots tel que $\ell(M) = n$. Pour tout mot m, on a :

— soit $m = \varepsilon$;

— soit $m = 0m'$ pour m' un autre mot ;

— soit $m = 1m'$ pour m' un autre mot.

On peut donc définir à partir de M trois ensembles $M_\varepsilon = M \cap \{\varepsilon\}$, $M_0 = \{m' \mid 0m' \in M\}$ et $M_1 = \{m' \mid 1m' \in M\}$. On a donc $M = M_\varepsilon \cup \{0m \mid m \in M_0\} \cup \{1m \mid m \in M_1\}$, ces unions étant disjointes.

On remarque que la longueur des mots de M_0 et M_1 est au plus $n-1$, puisque la longueur des mots de M est au plus n. Ainsi, par hypothèse de récurrence forte, il existe un unique arbre de mots réduit a_0 représentant M_0 et un unique arbre de mots réduit a_1 représentant M_1. En posant $b = T$ si $M_\varepsilon = \{\varepsilon\}$ et $b = F$ sinon, on a donc $N(b, a_0, a_1)$ un arbre de mots représentant M. Cet arbre de mot n'est pas réduit, car soit M_ε est non vide (donc $b = T$), soit M_0 ou M_1 est non vide (donc a_0 ou a_1 n'est pas l'arbre vide).

Pour l'unicité, soit a un arbre de mot réduit représentant M. On a donc $a \neq V$ car $M \neq \emptyset$, donc $a = N(b, g, d)$. Or, $b = T$ si et seulement si $\varepsilon \in M$, et g et d sont des arbres de mots réduits représentant exactement l'ensemble des mots de M privés du préfixe 0 ou 1, c'est-à-dire $\mathbb{M}(g) = M_0$ et $\mathbb{M}(d) = M_1$. Par hypothèse de récurrence forte, $g = a_0$ et $d = a_1$ donc $a = N(b, a_0, a_1)$ est l'unique arbre de mots réduit représentant M.

Ainsi, pour tout ensemble fini de mots M, il existe un unique arbre de mots réduit représentant M.

Dans la suite, on note $\mathbb{A}(M)$ l'unique arbre de mots réduit représentant l'ensemble de mots M. On a donc $\mathbb{M}(\mathbb{A}(M)) = M$ pour tout ensemble de mots M et $\mathbb{A}(\mathbb{M}(a)) = a$ pour tout arbre de mots a.

▶ **Question 30** Écrire une fonction `zeros_puis_uns : int -> arbre` qui prend en argument un entier $n \geq 0$ et qui renvoie l'arbre de mots réduit représentant l'ensemble des mots de la forme 0^p1^q avec $p + q = n$. *Indication : l'arbre a_3 de la Figure 1 est l'arbre* `zeros_puis_uns 3`.

▶ **Question 31** Écrire une fonction `sN : arbre -> arbre` qui prend en argument un arbre de mots a et qui renvoie V si $a = N(\mathrm{F}, V, V)$ et qui renvoie a sinon.

▶ **Question 32** Écrire une fonction `compter : arbre -> int` qui prend en argument un arbre de mots a et qui renvoie le cardinal de $\mathbb{M}(a)$. On attend une complexité en $\mathcal{O}(t(a))$ sans avoir besoin de la justifier.

▶ **Question 33** Écrire une fonction `recherche : mot -> arbre -> bool` qui prend en argument un mot m et un arbre de mots a et qui renvoie `true` si $m \in \mathbb{M}(a)$, `false` sinon. On attend une complexité en $\mathcal{O}(\ell(m))$ sans avoir besoin de la justifier.

▶ **Question 34** Écrire une fonction `ajouter : mot -> arbre -> arbre` qui prend en argument un mot m et un arbre de mots a et qui renvoie l'arbre de mots réduit représentant l'ensemble de mots $\mathbb{M}(a) \cup m$. Déterminer la complexité de cette fonction.

Solution La fonction effectue exactement un appel récursif par caractère de m et le coût d'un appel hors appel récursif est en $\mathcal{O}(1)$ (filtrage, test, application d'un constructeur). Ainsi, la complexité de la fonction est en $\mathcal{O}(\ell(m))$.

▶ **Question 35** Écrire une fonction `supprimer : mot -> arbre -> arbre` qui prend en argument un mot m et un arbre de mots a et qui renvoie l'arbre de mots réduit représentant l'ensemble de mots $\mathbb{M}(a) \setminus \{m\}$. Sans justifier, donner la complexité de cette fonction.

Solution La complexité est également en $\mathcal{O}(\ell(m))$.

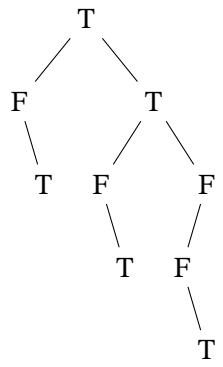
Dans la suite, on remarque que pour un ensemble de mots M, si on pose a' l'arbre de mots $\mathbb{A}(M)$ dans lequel on remplace toutes les étiquettes F par T, alors $\mathbb{M}(a') = \{m[:j] \mid m \in M \text{ et } 0 \leq j \leq \ell(m)\}$ l'ensemble des préfixes d'au moins un mot de M.

II.C Arbres des suffixes

Dans cette partie, pour un mot m, on définit l'**arbre des suffixes de** m, noté $\mathbb{AS}(m)$, comme l'arbre de mots réduit représentant l'ensemble des suffixes de m, c'est-à-dire $\mathbb{AS}(m) = \mathbb{A}(S(\{m\}))$.

▶ **Question 36** Déterminer l'arbre des suffixes de $m = 1101$.

Solution



▶ **Question 37** Déterminer en la justifiant $h(\mathbb{AS}(m))$ pour tout mot m. En déduire une fonction `retrouver_mot : arbre -> mot` qui retrouve le mot m à partir de l'arbre $\mathbb{AS}(m)$ en entrée de complexité linéaire en la taille de cet arbre. Il n'est pas demandé de la justifier.

Solution Le mot m est le plus long suffixe de m. Or, les feuilles de l'arbre $\mathbb{AS}(m)$ ont tous une étiquette T puisqu'il est réduit et correspondent donc à des suffixes de m, et la profondeur de cette feuille est égale à la longueur du suffixe correspondant. Ainsi, la hauteur de l'arbre $\mathbb{AS}(m)$ est égale à la longueur du mot m : $\underline{h(\mathbb{AS}(m)) = \ell(m)}$.

▶ **Question 38** Montrer que pour un mot m, l'arbre a' obtenu en remplaçant toutes les étiquettes des nœuds de $\mathbb{AS}(m)$ par `true` est l'arbre $\mathbb{A}(F(m))$.

Solution Considérons un nœud n de l'arbre $\mathbb{AS}(m)$. Puisque l'arbre est réduit, l'arbre enraciné en n contient une feuille $N(\mathrm{T}, V, V)$. Le chemin de la racine de $\mathbb{AS}(m)$ à n fait donc partie du chemin de la racine de $\mathbb{AS}(m)$ à cette feuille, qui correspond à un mot $s \in S(\{m\})$. En arrêtant ce mot à la profondeur de n, on obtient un préfixe de s, c'est-à-dire un préfixe d'un suffixe de m, donc un facteur de m. On a donc $\underline{\mathbb{M}(a') \subseteq F(m)}$.
Pour tout facteur $f = m[i : j]$, considérons $s = m[i :]$ qui est représenté par un nœud n d'étiquette T dans l'arbre a. Dans le chemin de la racine de a' à n, le nœud de profondeur $j - i$ est étiqueté T puisque s est un suffixe de m et représente bien le mot f. Ainsi, $f \in \mathbb{M}(a')$ et donc $\underline{F(m) \subseteq \mathbb{M}(a')}$.
Ainsi, $\underline{\mathbb{M}(a') = F(m)}$.

▶ **Question 39** En déduire que la taille de l'arbre $\mathbb{AS}(m)$ est au moins linéaire et au plus quadratique en $\ell(m)$. Montrer que ces bornes sont atteintes, en exhibant pour tout $\ell \in \mathbb{N}$ un mot m tel que $\mathbb{AS}(m)$ ait une taille $\ell(m) + 1$, et un mot m' tel que $\mathbb{AS}(m')$ ait une taille au moins égale à $C\ell^2$ pour une constante C à déterminer. *Pour la borne supérieure, on pourra se restreindre aux longueurs paires.*

Solution Soit $n \in \mathbb{N}$.
Un mot de longueur ℓ admet au moins un facteur de longueur k pour tout $k \in [\![0, \ell]\!]$. Puisque chaque facteur est représenté par un nœud de l'arbre $\mathbb{AS}(m)$, on a $\underline{t(\mathbb{AS}(m)) \geq \ell(m)+1}$. Pour le mot $\underline{m = 0^\ell}$, pour tout $k \in [\![0, \ell]\!]$, le facteur 0^k est le seul suffixe de m et $\mathbb{AS}(m)$ est donc un arbre peigne gauche de hauteur ℓ et de taille $\underline{\ell + 1 = |F(M)|}$.
Les facteurs non vides de m sont de la forme $m[i : j]$ avec $0 \leq i < j \leq \ell(m)$. Le nombre de facteurs distincts de m est donc majoré par le nombre de possibilités pour i et j plus un pour le facteur vide, ce qui donne $\frac{\ell(m)(\ell(m)+1)}{2} + 1$ qui est bien $\underline{\text{quadratique en } \ell(m)}$.

Soit $m = 0^p 1^p$ avec $p \in \mathbb{N}$. Alors :
- Les facteurs de longueur $k \in [\![0, p]\!]$ sont de la forme $0^j 1^{k-j}$ pour $j \in [\![0, k]\!]$ donc il y en a $k+1$.
- Les facteurs de longueur $k \in [\![p+1, 2p]\!]$ avec $\ell(m) = 2p$ sont de la forme $0^{j+k-p}1^{p-j}$ avec $j \in [\![0, 2p-k]\!]$, puisqu'un facteur de longueur $k > p$ contient au moins $k-p$ zéros suivis d'au moins $k-p$ uns. Il y a donc $2p-k+1$ facteurs de longueur k.

Ainsi, on a

$$\begin{aligned}
|F(m)| &= \sum_{k=0}^{p}(k+1) + \sum_{k=p+1}^{2p}(2p-k+1) \\
&= \frac{(p+1)(p+2)}{2} + \sum_{k=p+1}^{2p}(p+1+p-k) \\
&= \frac{(p+1)(p+2)}{2} + p(p+1) - \sum_{k'=1}^{p} k' \\
&= \frac{(p+1)(p+2)}{2} + p(p+1) - \frac{p(p+1)}{2} \\
&= (p+1)\left(\frac{p+2}{2} + p - \frac{p}{2}\right) \\
&= (p+1)\left(\frac{p}{2} + 1\right) \\
&\geq \frac{1}{2}p^2 = \underline{\frac{1}{4}\ell(m)^2}.
\end{aligned}$$

Ainsi, pour toute longueur ℓ paire, il existe un mot m' de longueur ℓ tel que $t(\mathbb{AS}(m')) \geq \frac{1}{4}\ell^2$.

Le second problème dans la Section II est tiré de l'épreuve d'option informatique A des écrits de l'ENS 2021. Les élèves intéressés y trouveront une représentation compacte de l'arbre des facteurs d'un mot binaire, permettant notamment de calculer en temps linéaire en $\ell(m)$ la longueur du plus long facteur répété de m, ou encore une méthode pour construire les mots de taille ℓ comprenant le plus grand nombre de facteurs distincts en utilisant les graphes eulériens.