Devoir en temps limité du Vendredi 9 Juin

Dans l'ensemble du devoir, toutes les fonctions seront :
- *écrites en langage OCaml,*
- *précédées d'une explication des variables utilisées,*
- *précédées d'une explication de l'algorithme.*

Exercice 1 (Du cours)

1. Une implémentation en OCaml de la structure de pile d'entier est le type enregistrement suivant :

```
type pile = {mutable contenu : int list} ;;
```

   (a) Définir dans cette situation les primitives d'accès `creer_pile`, `etre_pile_vide`, `empiler` et `depiler`.

   (b) Écrire en OCaml une fonction `insere_pile x p` ayant pour signature

```
insere_pile : int -> pile -> pile
```

   qui place un entier `x` à la bonne place dans une pile `p` triée.

   (c) Écrire en OCaml une fonction `tri_insertion_pile p` ayant pour signature

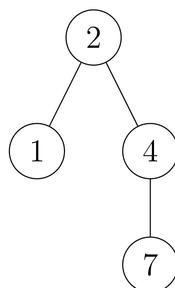
```
tri_insertion_pile : pile -> pile
```

   qui trie une pile `p` en suivant le principe du tri par insertion.

2. Une implémentation en OCaml d'un type d'arbre binaire homogène est le type personnalisé suivant :

```
type 'a arbre =
  | Vide
  | N of 'a * 'a arbre * 'a arbre
;;
```

   (a) Donner les instructions pour définir l'arbre suivant sur OCaml :



   (b) Définir en OCaml une fonction récursive `noeuds` ayant pour signature

```
noeuds : 'a arbre -> int
```

   qui calcule le nombre de noeuds d'un arbre binaire homogène.

   (c) Définir en OCaml une fonction récursive `hauteur` ayant pour signature

```
hauteur : 'a arbre -> int
```

   qui calcule la hauteur d'un arbre binaire homogène.

(d) En testant les deux fonctions précédentes sur différents exemples, on remarque que, pour tout arbre binaire non vide à n noeuds et de hauteur h, on a :

$$h \leqslant n \leqslant 2^h - 1.$$

Démontrer cette propriété par induction structurelle.

(e) Rappeler la définition d'un parcours en profondeur en mode préfixe d'un arbre. On pourra donner un exemple pour illustrer cette définition.

(f) Définir en OCaml une fonction récursive `prefixe` ayant pour signature

```
prefixe : 'a arbre -> 'a list
```

qui renvoie la liste du traitement des étiquettes des sommets dans le cas d'un parcours en profondeur en mode préfixe d'un arbre binaire homogène.

Exercice 2 (Extrait d'un sujet de concours)

Nous nous intéressons ici au problème de la satisfiabilité des formules booléennes, appelé SAT dans la littérature. Historiquement, SAT a joué un rôle prépondérant dans le développement de la théorie de la complexité. De nos jours, il intervient dans de nombreux domaines de l'informatique où des problèmes combinatoires difficiles apparaissent, comme la vérification formelle, la recherche opérationnelle, la bioinformatique, la cryptologie, l'apprentissage automatique, la fouille de données, et bien d'autres encore. Signe de son importance, SAT et ses variantes ont leur propre conférence internationale qui se tient tous les ans depuis près de 20 ans.

Ce sujet concerne une version restreinte de SAT appelée k-SAT, dans laquelle les formules considérées en entrée sont en forme normale conjonctive avec au plus k littéraux par clauses. Le sujet s'articule autour des différentes valeurs de k : $k = 1$ pour la partie I, $k = 2$ pour la partie II, $k \geqslant 3$ pour la partie III. La partie IV fait le lien avec le problème SAT lui même.

Préliminaires

Cette partie préliminaire introduit formellement les concepts et résultats utiles pour l'analyse.

Par complexité (en temps) d'un algorithme A, on entend le nombre d'opérations élémentaires nécessaires à l'exécution de A dans le cas le pire. Lorsque ce nombre dépend d'un ou plusieurs paramètres $k_0, \dots, k_{r-1}$, on dit que A a un temps d'exécution $\mathcal{O}(f(k_0, \dots, k_{n-1}))$ s'il existe une constante $C > 0$ telle que pour toutes les valeurs des k_i assez grandes et pour toute instance du problème de paramètres $k_0, \dots, k_{n-1}$, le nombre d'opérations élémentaires est au plus $Cf(k_0, \dots, k_{r-1})$. On parle de complexité *linéaire* (resp. *polynomiale*) quand f est une fonction linéaire (resp. un polynôme des n variables). Sauf mention contraire de l'énoncé, le candidat n'a pas à justifier la complexité des algorithmes. Toutefois, il devra veiller à ce que cette complexité ne dépasse pas les bornes prescrites.

Formules booléennes. Une *variable booléenne* est une variable prenant ses valeurs dans l'ensemble {`Vrai`, `Faux`}. Une *formule booléenne* s'obtient en combinant des variables booléennes et des connecteurs logiques Et (noté $\wedge$), Ou (noté $\vee$), Non (noté $\neg$) selon la grammaire suivante, donnée directement dans le langage OCaml :

```
type formule =
  | Var of int
  | Non of formule
  | Et of formule * formule
  | Ou of formule * formule
;;
```

Ainsi, les formules booléennes sont représentées par des structures arborescentes en machine, appelées *arbres d'expression* dans la suite. Voici la figure 1 pour un exemple.

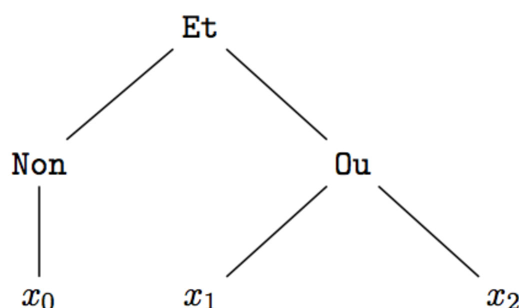


Figure 1 - L'arbre d'expression associé à la formule $(\neg x_0) \wedge (x_1 \vee x_2)$

A noter que les variables d'une formule f sont indexées par des entiers, d'où la ligne `Var of int` dans la définition du type `formule`. Par défaut, ces entiers seront supposés positifs ou nuls, contigus, et commençant par 0.
Ainsi, les variables de f seront dénotées par exemple $x_0, \dots, x_{r-1}$. La *taille* de f est le nombre total de variables booléennes et de connecteurs logiques qui la composent. C'est donc le nombre total n de sommets composant l'arbre d'expression associé, et on a naturellement $r \leqslant n$.

Valuations et équivalence logique. Étant données r variables booléennes $x_0, \dots, x_{r-1}$, une *valuation* est une application $\sigma : \{x_0, \dots, x_{r-1}\} \to \{\texttt{Vrai}, \texttt{Faux}\}$. Étant donnée une formule f utilisant ces r variables, le résultat de l'évaluation de f sur σ, noté $\sigma(f)$, est obtenu en affectant la valeur $\sigma(x_i)$ à chaque variable x_i.
Deux formules f et g utilisant les variables booléennes $x_0, \dots, x_{r-1}$ sont dites *logiquement équivalentes* si $\sigma(f) = \sigma(g)$ pour toute valuation $\sigma : \{x_0, \dots, x_{r-1}\} \to \{\texttt{Vrai}, \texttt{Faux}\}$.

Propriétés des connecteurs logiques. Rappelons que le connecteur $\neg$ est involutif, c'est-à-dire que pour toute formule f

- $\neg\neg f$ est logiquement équivalente à f.

Par ailleurs, en plus d'être commutatifs et associatifs, les connecteurs logiques $\wedge$ et $\vee$ sont distributifs, c'est-à-dire que pour toutes formule f, g et h

- $f \wedge (g \vee h)$ est logiquement équivalente à $(f \wedge g) \vee (f \wedge h)$,
- $f \vee (g \wedge h)$ est logiquement équivalente à $(f \vee g) \wedge (f \vee h)$.

Dans ce sujet nous adoptons la convention que le connecteur $\neg$ est prioritaire sur $\wedge$ et $\vee$, ce qui permet par exemple d'écrire $\neg x_0 \wedge (x_1 \vee x_2)$ au lieu de $(\neg x_0) \wedge (x_1 \vee x_2)$ comme dans la figure 1.
Les *lois de De Morgan* décrivent la manière dont $\wedge$ et $\vee$ interagissent avec $\neg$. Pour toutes formules f et g :

- $\neg(f \vee g)$ est logiquement équivalente à $\neg f \wedge \neg g$,
- $\neg(f \wedge g)$ est logiquement équivalente à $\neg f \vee \neg g$.

Le problème SAT. Étant donnée une formule f à r variables $x_0, \ldots, x_{r-1}$, le problème SAT consiste à déterminer s'il existe une valuation $\sigma : \{x_0, \ldots, x_{r-1}\} \to \{\texttt{Vrai}, \texttt{Faux}\}$ telle que $\sigma(f) = \texttt{Vrai}$. Dans l'affirmative, la formule f est dite *satisfiable*. Dans la négative, f est dite *insatisfiable*. Par exemple, la formule $\neg x_0 \wedge (x_1 \vee x_2)$ de la figure 1 est satisfiable, tandis que $\neg x_0 \wedge (x_1 \vee x_2) \wedge (x_0 \vee \neg x_1) \wedge (x_0 \vee \neg x_2)$ est insatisfiable.

Q.1 Pour chaque formule qui suit, dire si elle est satisfiable ou non, sans justification :

a. $x_1 \wedge (x_0 \vee \neg x_0) \wedge \neg x_1$
b. $(x_0 \vee \neg x_1) \wedge (\neg x_0 \vee x_2) \wedge (x_1 \vee \neg x_2)$
c. $x_0 \wedge \neg(x_0 \wedge \neg(x_1 \wedge (x_1 \wedge \neg x_2)))$
d. $(x_0 \vee x_1) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_1)$

Forme normale conjonctive. Dans la suite nous utiliserons essentiellement des formules écrites sous la forme suivante, appelée *forme normale conjonctive* (FNC) :

$$\bigwedge_{i=1}^{m} \bigvee_{j=1}^{n_i} l_{i,j} \tag{1}$$

où chaque *littéral* $l_{i,j}$ est soit une variable booléenne x, soit sa négation $\neg x$, et où les littéraux sont regroupés en *clauses disjonctives* $\bigvee_{j=1}^{n_i} l_{i,j}$. Par exemple, les formules *a.*, *b.* et *d.* de la question 1 sont des FNC.

Une FNC est appelée k-FNC lorsque chaque clause a au plus k littéraux, c'est-à-dire que $n_i \leqslant k$ pour tout $i \in \{1, \ldots, m\}$ dans l'équation (1). Notez qu'alors la formule est aussi une k'-FNC pour tout $k' \geqslant k$. Par exemple, les formules *a.*, *b.* et *d.* de la question 1 sont des 2-FNC. La variante de SAT appelée k-SAT prend uniquement en entrée des formules en k-FNC. C'est cette variante qui nous intéresse dans ce sujet. Elle est équivalente à SAT du point de vue de la théorie de la complexité quand $k \geqslant 3$, comme nous le verrons dans la partie 4.
En machine nous représenterons les FNC sous la forme de listes de listes. Plus précisément, une FNC sera une liste de clauses et chaque clause sera une liste de littéraux :

```
type litteral =
  | V of int (* variable *)
  | NV of int (* négation de variable *)
;;

type clause = litteral list ;;

type fnc = clause list ;;
```

Ainsi, une formule en k-FNC sera représentée par une liste (de taille arbitraire) de listes de taille au plus k chacune.

Q.2 Écrire une fonction `var_max_clause : clause → int` qui prend en entrée une clause et renvoie le plus grand indice de variable utilisé dans la clause.
En déduire une fonction `var_max : fnc → int` qui prend en entrée une FNC f et renvoie le plus grand indice de variable utilisé dans la formule.

Partie 1 : Résolution de 1-SAT

Commençons par le cas le plus simple, à savoir $k = 1$. Ici chaque clause de la FNC est formée d'un unique littéral l_i et donc impose un unique choix possible d'affectation pour la variable x_i : soit $l_i = x_i$ et dans ce cas x_i doit valoir `Vrai`, soit $l_i = \neg x_i$ et dans ce cas x_i doit valoir `Faux`. La formule est alors satisfiable si et seulement s'il n'y a pas de contradiction dans les choix d'affectation de variables imposés par ses différentes clauses. Afin d'effectuer ce test efficacement, nous allons maintenir un tableau où chaque case correspondra à une variable de la formule (de même indice que la case) et où les valeurs sont des *triléens* : vrai, faux ou indéterminé. Pour cela, nous définissons le type `trileen` ci-dessous :

```
type trileen =
  | Vrai
  | Faux
  | Indetermine
;;
```

Grâce au tableau de triléens, à chaque littéral rencontré, on peut déterminer en temps constant si la variable x_i est déjà affectée ou non, et dans l'affirmative, si sa valeur d'affectation est compatible avec celle imposée par l_i.

Q.3 Écrire une fonction `un_sat : fnc → bool` qui prend en entrée une FNC f, supposée être une 1-FNC, et qui renvoie un booléen valant `true` si et seulement si f est satisfiable. La complexité de la fonction doit être linéaire en la taille de f.

Partie 2 : Résolution de 2-SAT

Nous venons de voir que 1-SAT est un problème facile puisque résoluble en temps linéaire. Nous allons maintenant voir que 2-SAT est également linéaire, bien que son traitement efficace nécessite plus de travail d'analyse et de codage. Nous allons en effet montrer comment réduire les instances de 2-SAT à la recherche de composantes fortement connexes dans un graphe orienté.
Soit G un graphe orienté. Un tel objet est défini par deux ensembles finis : l'ensemble V des sommets (ou noeuds) et l'ensemble E des arêtes orientées (ou arcs orientés). Chaque arête de E relie deux sommets dans un ordre précis. C'est donc un élément de $V \times V$. La *taille* du graphe G est $|V| + |E|$ où $|V|$ et $|E|$ désignent respectivement le nombre de sommets et le nombre d'arêtes.
Par défaut les sommets de G sont indexés par les entiers 0 à $|V|-1$ inclus. Ainsi, on pourra les désigner par $v_0, v_1, \ldots, v_{|V|-1}$. En machine nous représentons G par *listes d'adjacences*, plus précisément par un tableau de listes d'entiers :

```
type graphe = int list array ;;
```

où les indices du tableau correspondent à ceux des noeuds du graphe et où la liste associée à la case d'indice i dans le tableau contient les indices des *successeurs* du noeud v_i dans le graphe, c'est-à-dire les entiers j tels que $(v_i, v_j) \in E$. Voir la figure 2 pour une illustration :

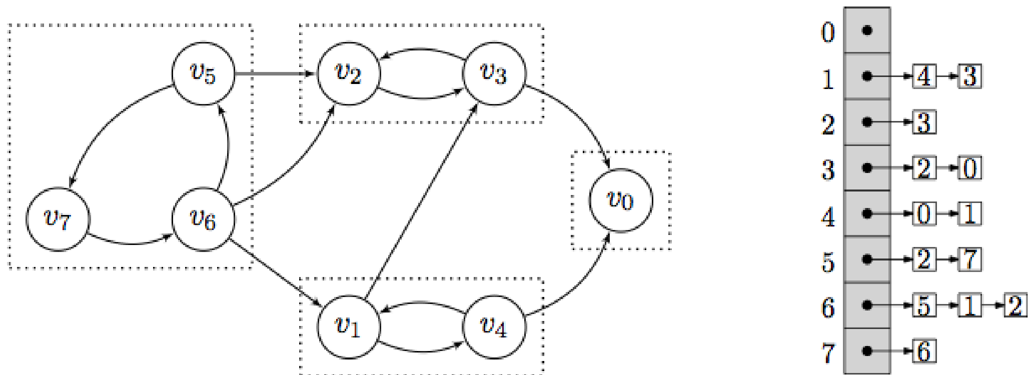


Figure 2 - Un exemple de graphe orienté. Les composantes fortement connexes du graphe sont encadrées en pointillés.

Un *chemin* d'un sommet s à un sommet t dans G est une suite finie de sommets $(s = v_{i_0}, v_{i_1}, \ldots, v_{i_{k-1}}, v_{i_k} = t)$ telle que $(v_{i_l}, v_{i_{l+1}}) \in E$ pour tout $l = 0, 1, \ldots, k-1$. Ici, k désigne la *longueur* du chemin. Par exemple, le graphe de la figure 2 contient le chemin $(v_6, v_5, v_7, v_6, v_1, v_3, v_2)$ de longueur 6 (on remarque que ce chemin contient un *circuit* c'est-à-dire un chemin dont les deux sommets extrémités sont identiques) et le chemin (v_0) de longueur nulle, mais pas le chemin (v_0, v_4) de longueur 1.
Une *composante fortement connexe* de G est un sous-ensemble S de ses sommets, maximal pour l'inclusion, tel que pour tout couple de sommets $(s, t) \in S^2$, il existe un chemin de s à t dans G. Voir la figure 2 pour une illustration.

Nous admettons pour la suite avoir écrit une fonction `cfc : graphe → int list list` qui prend en entrée un graphe et qui renvoie une liste de listes d'entiers contenant l'ensemble des composantes fortement connexes du graphe, chaque composante étant stockée dans l'une des listes d'entiers. Nous admettons également que la complexité de la fonction est linéaire en la taille du graphe.

La réduction d'une instance 2-SAT à un calcul de composantes fortement connexes dans un graphe repose sur l'observation simple que toute clause $(l_i \vee l_j)$ est logiquement équivalente à $(\neg l_i) \Rightarrow l_j$, elle même équivalente à sa contraposée $(\neg l_j) \Rightarrow l_i$. Ainsi

$$l_i \vee l_j \equiv (\neg l_j \Rightarrow l_i) \wedge (\neg l_i \Rightarrow l_j)$$

Lorsqu'une clause est formée d'un seul littéral l_i, il suffit de l'exprimer sous la forme équivalente $(l_i \vee l_i)$, ce qui donne $(\neg l_i) \Rightarrow l_i$, qui est sa propre contraposée.
Cette observation suggère la procédure suivante pour construire un graphe orienté G à partir d'une 2-FNC f à r variables (notées $x_0, \ldots, x_{r-1}$) :

- Pour chaque variable x_i on ajoute les sommets v_{2i} et v_{2i+1} à G, qui représentent respectivement le littéral x_i et le littéral $\neg x_i$.
  Les arêtes du graphe (les "flèches") représentent les implications logiques correspondant à la formule f. Autrement dit :

- Pour chaque clause de type (l_i) présente dans la formule f, on ajoute une arête à G soit du sommet v_{2i+1} au sommet v_{2i} si $l_i = x_i$, soit du sommet v_{2i} au sommet v_{2i+1} si $l_i = \neg x_i$.
- Pour chaque clause de la formule f du type $(l_i \vee l_j)$ où les littéraux l_i, l_j contiennent des variables x_i, x_j distinctes, on ajoute deux arêtes à G, choisies en fonction des cas suivants :
  - si $l_i = x_i$ et $l_j = x_j$ alors on ajoute les arêtes (v_{2i+1}, v_{2j}) et (v_{2j+1}, v_{2i}),
  - si $l_i = x_i$ et $l_j = \neg x_j$ alors on ajoute les arêtes (v_{2i+1}, v_{2j+1}) et (v_{2j}, v_{2i}),
  - si $l_i = \neg x_i$ et $l_j = x_j$ alors on ajoute les arêtes (v_{2i}, v_{2j}) et (v_{2j+1}, v_{2i+1}),
  - si $l_i = \neg x_i$ et $l_j = \neg x_j$ alors on ajoute les arêtes (v_{2i}, v_{2j+1}) et (v_{2j}, v_{2i+1}).
- Enfin, pour chaque clause du type $(l_i \vee l_j)$ où les littéraux l_i, l_j contiennent la même variable $x_i = x_j$, soit on élimine directement la clause si elle est de la forme $(x_i \vee \neg x_i)$ ou $(\neg x_i, x_i)$, soit on se ramène au cas d'une clause (l_i) sinon.

Par exemple, sur la formule $x_0 \wedge (x_1 \vee \neg x_0)$, la procédure donne un graphe avec quatre sommets : v_0, v_1, v_2, v_3 et trois arêtes : (v_1, v_0), (v_3, v_1), (v_0, v_2), comme illustré dans la figure 3 :

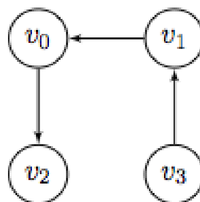


Figure 3 - Le graphe obtenu à partir de la formule $x_0 \wedge (x_1 \vee \neg x_0)$

On remarque, par transitivité de l'implication que s'il existe un chemin entre deux sommets v_i et v_j, alors $f \models (v_i \implies v_j)$.

Q.4 Donner le résultat de la procédure ci-dessus sur la formule :

$$(x_1 \vee x_2) \wedge x_0 \wedge (x_2 \vee \neg x_2) \wedge (\neg x_2 \vee x_0).$$

Q.5 Écrire une fonction `deux_sat_vers_graphe : fnc → graphe` qui prend en argument une FNC f, supposée être une 2-FNC, et qui renvoie le graphe G obtenu par la procédure ci-dessus. On pourra utiliser la fonction `var_max` codée à la question 2. Pour simplifier, on supposera qu'aucune clause n'est répétée dans f et qu'aucune clause n'est de type $(l_i \vee l_j)$ où l_i, l_j contiennent la même variable $x_i = x_j$. La complexité de la fonction doit être linéaire en la taille de f.

On pourra utiliser la fonction `List.iter` *telle que* `List.iter ajout_clause f` *applique la fonction* `ajout_clause` *de type* `clause -> unit()` *à tous les éléments de la 2-FNC* `f`.

Q.6 Supposons que la 2-FNC initiale f soit satisfiable et soit σ une valuation telle que $\sigma(f) =$ `Vrai`. Montrer qu'alors, pour tous sommets v_i, v_j situés dans une même composante fortement connexe de G, les variables booléennes $x_{\lfloor i/2 \rfloor}$ et $x_{\lfloor j/2 \rfloor}$ correspondantes vérifient $\sigma(x_{\lfloor i/2 \rfloor}) = \sigma(x_{\lfloor j/2 \rfloor})$ si $(i - j)$ est pair et $\sigma(x_{\lfloor i/2 \rfloor}) = \neg\sigma(x_{\lfloor j/2 \rfloor})$ si $i - j$ est impair. Ici, $\lfloor . \rfloor$ désigne la partie entière inférieure.

Q.7 En déduire que si f est satisfiable, alors il n'existe pas de variable x_i dont les deux sommets correspondants v_{2i} et v_{2i+1} soient dans la même composante fortement connexe du graphe G.

Réciproquement, on peut montrer que s'il n'existe pas de variable x_i de f dont les deux sommets correspondants v_{2i} et v_{2i+1} soient dans la même composante fortement connexe du graphe G, alors en attribuant la même valeur booléenne à tous les littéraux impliqués dans chacune des composantes fortement connexes de G, on construit une valuation qui satisfait f. Ceci fournit un critère simple pour décider de la satisfiabilité de f.

Q.8 Écrire une fonction `deux_sat : fnc → bool` qui prend en entrée une FNC f supposée être une 2-FNC, et qui renvoie un booléen indiquant si f est satisfiable ou non. La complexité de la fonction doit être linéaire en la taille de f.
On pourra utiliser la fonction `List.iter`

Partie 3 : Résolution de k-SAT pour k arbitraire

Nous allons maintenant décrire un algorithme pour la résolution de k-SAT dans le cas général. Le principe de base de l'algorithme est de faire une recherche exhaustive sur l'ensemble des valuations possibles des variables de la formule. Pour chaque valuation considérée, on évalue la formule : si le résultat est `Vrai` alors on renvoie `Faux`, si le résultat est `Faux` alors on rejette la valuation courante et on passe à la suivante. L'algorithme est en fait un peu plus malin que cela : il évalue la formule également pour des valuations partielles et décide, soit d'accepter la valuation partielle courante si le résultat de l'évaluation est déjà `Vrai`, soit de la rejeter précocement si le résultat est déjà `Faux`, soit enfin de compléter la construction de la valuation si le résultat de l'évaluation est encore `Indetermine`.

Pour coder les valuations partielles en machine, nous allons utiliser des tableaux de triléens. Rappelons que le type `trileen` a été introduit dans la partie 1. Ce type nous fait travailler non plus dans l'algèbre de Boole, où les variables prennent leurs valeurs parmi les deux booléens habituels, mais dans l'algèbre ternaire dite de Kleene, où les variables prennent leurs valeurs parmi les trois triléens. Les nouvelles tables de vérité des connecteurs $\wedge$, $\vee$ et $\neg$ sont données ci-dessous.

$a \wedge b$		b = Vrai	b = Indét.	b = Faux
a	Vrai	Vrai	Indét.	Faux
	Indét.	Indét.	Indét.	Faux
	Faux	Faux	Faux	Faux

$a \vee b$		b = Vrai	b = Indét.	b = Faux
a	Vrai	Vrai	Vrai	Vrai
	Indét.	Vrai	Indét.	Indét.
	Faux	Vrai	Indét.	Faux

a	$\neg a$
Vrai	Faux
Indét.	Indét.
Faux	Vrai

Figure 4 - Tables de vérité des connecteurs logiques usuels sur les triléens.

Q.9 Écrire les trois fonctions `et : trileen → trileen → trileen`, `ou : trileen → trileen → trileen`, `non : trileen → trileen` qui codent respectivement les connecteurs logiques $\wedge$, $\vee$, $\neg$ sur les triléens. La complexité de chaque fonction doit être constante.

Supposons maintenant que les variables d'une FNC prennent leurs valeurs parmi les triléens. Une récurrence immédiate montre alors qu'une clause disjonctive de la formule vaut `Vrai` quand l'un au moins de ses littéraux vaut `Vrai`, `Faux` quand tous ses littéraux valent `Faux` et `Indetermine` dans tous les autres cas. Une autre récurrence immédiate montre que la FNC elle même vaut `Vrai` quand toutes ses clauses valent `Vrai`, `Faux` quand au moins l'une de ses clauses vaut `Faux` et `Indetermine` dans tous les autres cas.

Q.10 Écrire une fonction `eval : fnc → trileen array → trileen` qui prend en entrée une FNC f ainsi qu'un tableau de triléens t, et qui renvoie un triléen indiquant si le résultat de l'évaluation de f sur la valuation partielle fournie dans t est `Vrai`, `Faux` ou `Indetermine`. On supposera que t a la bonne taille. La complexité de la fonction doit être linéaire en la taille de la formule.

Nous pouvons maintenant décrire l'algorithme de recherche exhaustive avec terminaison précoce. Pour itérer sur l'ensemble des valuations nous utilisons une approche récursive consistant à parcourir en profondeur les branches d'un arbre binaire sans le construire explicitement. Chaque niveau i de l'arbre correspond à l'affectation de la variable x_i, comme illustré dans la figure 5 pour le cas de 3 variables.

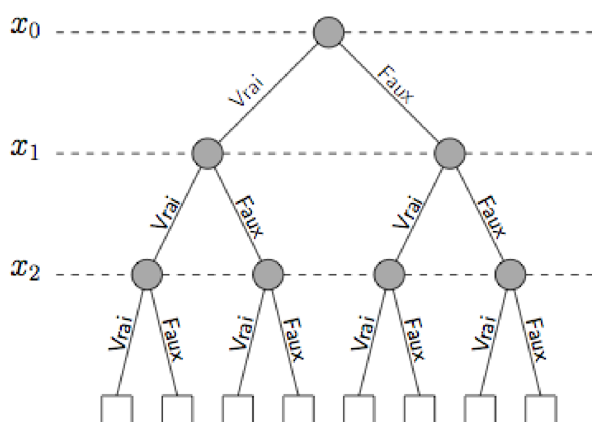


Figure 5 - Arbre parcouru lors de la recherche exhaustive parmi les valuations des variables x_0, x_1, x_2.

Au départ la valeur `Indetermine` est affectée à toutes les variables. Le parcours commence à la racine. À chaque noeud de l'arbre visité, avant toute affectation de la variable correspondante, un appel à la fonction `eval` est fait pour tester si le résultat de l'évaluation est :

- `Vrai`, auquel cas l'exploration s'arrête et la formule est satisfiable,
- `Faux`, auquel cas l'exploration de la branche courante de l'arbre s'interrompt prématurément pour reprendre au niveau du parent du noeud courant,
- `Indetermine`, auquel cas l'exploration de la branche courante de l'arbre se poursuit normalement.

Comme indiqué précédemment, pour stocker la valuation partielle courante on utilise un tableau de triléens dans lequel les variables non encore affectées prennent la valeur `Indetermine`.

Q.11 Écrire une fonction `k_sat : fnc → bool` qui prend en entrée une FNC f et qui renvoie un booléen valant `true` si f est satisfiable et `false` sinon. La fonction doit coder la méthode de recherche exhaustive avec terminaison précoce décrite ci-dessus. Un soin particulier doit être apporté à la clarté du code, dans lequel il est recommandé d'insérer des commentaires aux points clés.

Partie 4 : De k-SAT à SAT

Dès le début du sujet nous avons laissé de coté le problème SAT au profit de sa variante k-SAT. Comme toute instance du deuxième problème est également une instance du premier, k-SAT est a priori une version restreinte de SAT. En fait, il n'en est rien car, comme nous allons le voir dans cette partie, toute instance de SAT peut être transformée en une instance de k-SAT (pour $k \geqslant 3$) par un algorithme de complexité polynomiale. Ainsi, l'algorithme codé à la question 11, ou tout autre algorithme exponentiel optimisé pour k-SAT, peut en fait résoudre n'importe quelle instance de SAT avec la même complexité.

Pour la transformation proprement dite, la première étape consiste à mettre en FNC la formule booléenne considérée. En effet, toute formule booléenne peut être mise en FNC et une approche évidente pour ce faire est d'utiliser les propriétés des connecteurs logiques rappelées dans la partie préliminaire.

Q.12 Pour chacune des formules suivantes, utiliser l'involutivité de la négation ($\neg(\neg P) \equiv P$) l'associativité et la distributivité des connecteurs $\wedge$ et $\vee$, ainsi que les lois de De Morgan pour transformer la formule en FNC. Seul le résultat du calcul est demandé.

a. $(x_1 \vee \neg x_0) \wedge \neg(x_4 \wedge \neg(x_3 \wedge x_2))$.

b. $(x_0 \wedge x_1) \vee (x_2 \wedge x_3) \vee (x_4 \wedge x_5)$.

L'exemple b) de la question 12 se généralise à des formules de taille arbitraire ce qui montre que l'approche ci-dessus n'est pas efficace puisque la FNC obtenue peut avoir une taille exponentielle en la taille n de la formule booléenne f de départ. Nous allons donc adopter une autre stratégie, qui sera d'introduire de nouvelles variables et de remplacer f par une autre formule f' qui est *équisatisfiable*, c'est-à-dire que f' est satisfiable si et seulement si f l'est. La formule f' sera en FNC et sa taille polynomiale en n. La procédure pour construire f' à partir de f fonction en deux temps :

1. On commence par appliquer les lois de De Morgan récursivement à l'arbre d'expression associé à f, de manière à faire descendre toutes les négations au niveau des noeuds parents des variables. Soit f^* la nouvelle formule ainsi obtenue, qui par construction est logiquement équivalente à f. Par exemple, si f est la formule a) de la question 12, alors $f^* = (x_1 \vee \neg x_0) \wedge (\neg x_4 \vee (x_3 \wedge x_2))$.
2. Ensuite, on applique récursivement les règles de réécriture suivantes à l'arbre d'expression de f^* :
   - si $f^* = \phi^* \wedge \psi^*$, alors on pose $f' = \phi' \wedge \psi'$, où ϕ' et ψ' sont les versions réécrites de ϕ^* et ψ^* respectivement,
   - si $f^* = \phi^* \vee \psi^*$, alors on introduit une nouvelle variable booléenne x dans la formule et on pose $f' = \bigwedge_{i=1}^{p} (\phi'_i \vee x) \wedge \bigwedge_{j=1}^{q} (\psi'_j \vee \neg x)$, où $\phi' = \bigwedge_{i=1}^{p} \phi'_i$ et $\psi' = \bigwedge_{j=1}^{q} \psi'_j$ sont les versions réécrites de ϕ^* et ψ^* respectivement.

   Par exemple, en reprenant la formule f^* obtenue dans l'exemple de l'étape 1, on a $f' = (x_1 \vee x_5) \wedge (\neg x_0 \vee \neg x_5) \wedge (\neg x_4 \vee x_6) \wedge (x_3 \vee \neg x_6) \wedge (x_2 \vee \neg x_6)$ en introduisant les nouvelles variables x_5 et x_6.

Q.13 Montrer que les formules f et f' sont équisatisfiables.

Q.14 Écrire une fonction `neg_en_bas : formule → formule` qui effectue l'étape 1 ci-dessus, c'est-à-dire qu'elle prend en argument une formule f et renvoie une autre formule f^* logiquement équivalente et dans laquelle tous les connecteurs $\neg$ ont des variables pour fils dans l'arbre d'expression. La fonction doit avoir une complexité linéaire en la taille de f.

On se donne à présent une nouvelle fonction `var_max_formule`, qui prend une `formule` en argument et qui renvoie le plus grand indice de variable utilisé dans la formule. La complexité de la fonction est linéaire en la taille de la formule.

Q.15 Écrire une fonction `formule_vers_fnc : formule → fnc` qui prend en argument la formule f^* obtenue à l'issue de l'étape 1 et qui renvoie la FNC f' construite à l'étape 2. La complexité de la fonction doit être polynomiale en la taille de f^*.
On pourra utiliser la fonction `List.map`.