ÉCOLE POLYTECHNIQUE — ÉCOLES NORMALES SUPÉRIEURES

CONCOURS D'ADMISSION 2016 **FILIÈRES MP SPECIALITÉ INFO**

COMPOSITION D'INFORMATIQUE – A – (XULCR)

(Durée : 4 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation sera **obligatoirement** `Caml`.

*
* *

Satisfiabilité des formules booléennes

Nous nous intéressons ici au problème bien connu 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 se concentre sur une version restreinte de SAT, appelée k-SAT, dans laquelle les formules booléennes considérées en entrée sont en forme normale conjonctive avec au plus k littéraux par clause. Le but est d'apporter aux candidats les rudiments de l'analyse de la complexité de k-SAT, en proposant des approches efficaces pour des valeurs spécifiques de k ainsi qu'une approche générale pour les valeurs de k arbitraires. Le sujet s'articule donc naturellement autour des differentes valeurs de k : $k = 1$ pour la partie I, $k = 2$ pour la partie II, $k \geq 3$ 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 (comparaison, addition, soustraction, multiplication, division, affectation, test, etc) nécessaires à l'exécution de A dans le cas le pire. Lorsque ce nombre dépend d'un ou plusieurs paramètres $\kappa_0, \cdots, \kappa_{r-1}$, on dit que A a un temps d'exécution en $\mathcal{O}(f(\kappa_0, \cdots, \kappa_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $\kappa_0, \cdots, \kappa_{r-1}$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $\kappa_0, \cdots, \kappa_{r-1}$, le nombre d'opérations élémentaires est au plus $C\, f(\kappa_0, \cdots, \kappa_{r-1})$. On dit que la complexité est *linéaire* quand f est une fonction linéaire des paramètres $\kappa_0, \cdots, \kappa_{r-1}$. De même, on dit que la

complexité est *polynomiale* quand f est une fonction polynomiale des paramètres. Sauf mention contraire dans l'énoncé, le candidat n'aura pas à justifier de la complexité de ses algorithmes. Toutefois il devra veiller à ce que cette complexité ne dépasse pas les bornes prescrites.

Pour la programmation proprement dite, en plus des fonctionnalités de base du langage Caml le candidat pourra utiliser les fonctions suivantes sans les programmer :

- `do_list: ('a -> unit) -> 'a list -> unit`
  `do_list f [a1; a2; ...; an]` équivaut à `begin f a1; f a2; ...; f an; () end`
- `map: ('a -> 'b) -> 'a list -> 'b list`
  `map f [a1; a2; ...; an]` renvoie la liste `[f a1; f a2; ...; f an]`
- `it_list: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
  `it_list f a [b1; ...; bn]` renvoie la valeur de `f (... (f (f a b1) b2) ...)`
- `list_it: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
  `list_it f [a1; ...; an] b` renvoie la valeur de `f a1 (f a2 (... (f an b) ...))`

Formules booléennes. Une *variable booléenne* est une variable prenant ses valeurs dans l'ensemble $\{\texttt{Vrai}, \texttt{Faux}\}$. 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 Caml :

```
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. Voir la figure 1 pour un exemple.

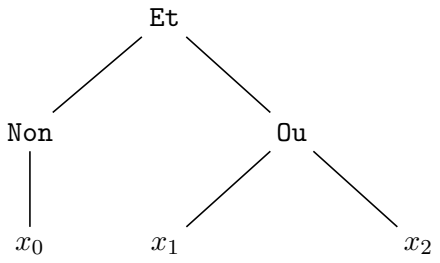

FIGURE 1 – L'arbre d'expression associé à la formule $(\neg x_0) \wedge (x_1 \vee x_2)$ de taille 6.

À 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 $x_0, \cdots, x_{r-1}$. 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 nœuds composant l'arbre d'expression associé, et on a naturellement $r \leq n$.

Valuations et équivalence logique. Étant données r variables booléennes $x_0, \cdots, x_{r-1}$, une *valuation* est une application $\sigma : \{x_0, \cdots, x_{r-1}\} \to \{\texttt{Vrai}, \texttt{Faux}\}$. 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 $x_0, \cdots, x_{r-1}$ sont dites *logiquement équivalentes* si $\sigma(f) = \sigma(g)$ pour toute valuation $\sigma : \{x_0, \cdots, x_{r-1}\} \to \{\texttt{Vrai}, \texttt{Faux}\}$.

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 formules 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 $x_0, \cdots, x_{r-1}$, le problème SAT consiste à déterminer s'il existe une valuation $\sigma : \{x_0, \cdots, x_{r-1}\} \to \{\texttt{Vrai}, \texttt{Faux}\}$ 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.

Question 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) $x_0 \wedge \neg(x_0 \wedge \neg(x_1 \wedge \neg(x_1 \wedge \neg x_2)))$
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) :

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

où chaque *littéral* l_{ij} est soit une variable booléenne x, soit sa négation $\neg x$, et où les littéraux sont regroupés en *clauses disjonctives* $\bigvee_{j=1}^{n_i} l_{ij}$. 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 $n_i \leq k$ pour tout $i \in \{1, \cdots, m\}$ dans l'équation (1). Notez qu'alors la formule est également

une k'-FNC pour tout $k' \geq k$. 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 $k \geq 3$, comme nous le verrons dans la partie IV.

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.

Question 2 Écrire une fonction `var_max` qui prend en entrée une FNC f et renvoie le plus grand indice de variable utilisé dans la formule. La complexité de la fonction doit être linéaire en la taille de f.

```
var_max: fnc -> int
```

Partie I. Résolution de 1-SAT

Commençons pas 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 seront 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 l_i 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.

Question 3 Écrire une fonction `un_sat` 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.

```
un_sat: fnc -> bool
```

Partie II. 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. Ce dernier problème a un intérêt en soi et fait l'objet de la sous-partie II.1. La réduction proprement dite sera détaillée dans la sous-partie II.2.

II.1 Recherche de composantes fortement connexes dans un graphe orienté

Soit G un graphe orienté. Rappelons qu'un tel objet est défini par deux ensembles finis : l'ensemble V des sommets (ou nœuds) 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 élement 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 $v_0, v_1, \cdots, v_{|V|-1}$. En machine nous représentons G par *listes d'adjacence*, plus précisément par un tableau de listes d'entiers :

```
type graphe == int list vect;;
```

où les indices du tableau correspondent à ceux des nœuds du graphe et où la liste associée à la case d'indice i dans le tableau contient les indices des *successeurs* du nœud 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.

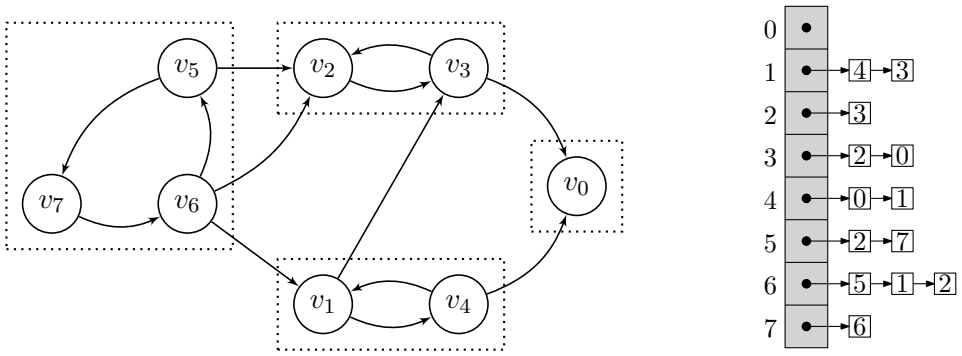

FIGURE 2 – Un exemple de graphe orienté avec, à droite, une représentation par listes d'adjacence. Les composantes fortement connexes du graphe sont encadrées en pointillés.

Rappelons qu'un *chemin* d'un sommet s à un sommet t dans G est une suite finie de sommets $(s = v_{i_0}, v_{i_1}, \cdots, v_{i_{k-1}}, v_{i_k} = t)$ telle que $(v_{i_l}, v_{i_{l+1}}) \in E$ pour tout $l = 0, 1, \cdots, k-1$. 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 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 partie gauche de la figure 2 pour une illustration. Le calcul des composantes

fortement connexes de G peut se faire naïvement en testant l'existence d'un chemin dans le graphe pour chaque couple de sommets. Cette approche revient à effectuer un parcours complet du graphe au départ de chaque sommet, par conséquent elle est trop coûteuse en temps. D'autres approches permettent de faire le même calcul en temps linéaire en la taille du graphe, dont celle de Kosaraju-Sharir que nous allons maintenant étudier. L'algorithme procède en deux étapes :

a) Il effectue un parcours en profondeur récursif de G et relève, pour chaque nœud visité v_i, l'instant t_i de fin de traitement du nœud. Cet instant se situe à la fin de l'exécution de la fonction de parcours sur le nœud v_i, après le retour des appels récursifs sur ses successeurs dans le graphe. Le code Caml correspondant est fourni ci-dessous : la fonction `dfs_tri` prend un `graphe` g en entrée et renvoie la liste des indices des sommets du graphe, ordonnée dans l'ordre inverse de leurs instants de fin de traitement (notez la ligne 8, qui insère l'indice du sommet courant en tête de liste au moment de la fin de son traitement par la fonction de parcours récursif `dfs_rec`) :

```
1  let dfs_tri g =
2    let deja_vu = make_vect (vect_length g) false in
3    let resultat = ref [] in
4    let rec dfs_rec i =
5      if not deja_vu.(i) then begin
6          deja_vu.(i) <- true;
7          do_list dfs_rec g.(i); (* voir page 2 pour la définition de do_list *)
8          resultat := i :: !resultat;
9      end in
10   for i = 0 to vect_length g - 1 do dfs_rec i done;
11   !resultat;;

dfs_tri: graphe -> int list
```

b) Il effectue un deuxième parcours en profondeur, cette fois du graphe G' obtenu en renversant le sens de toutes les arêtes de G. Les différents sommets de départ du parcours sont choisis non pas dans un ordre arbitraire comme à l'étape a) mais dans l'ordre décroissant des instants t_i. Pour chaque sommet de départ v_i, l'ensemble des sommets visités par le parcours en profondeur de G' à partir de v_i forme une composante fortement connexe du graphe initial G.

Illustrons l'algorithme sur l'exemple de la figure 2 :

a) Le parcours en profondeur du graphe initial G, en considérant les sommets de départ par ordre croissant d'indice comme dans la fonction `dfs_tri` (ligne 10), choisit d'abord v_0 comme point de départ. Le sous-graphe parcouru est alors le sommet v_0 seul. Ensuite, le parcours reprend au sommet v_1 et visite le sommet v_4, puis revient à v_1 et visite v_3 puis v_2, puis revient à v_3 puis à v_1 et s'arrête. Enfin, le parcours reprend au sommet v_5 et visite les sommets v_7 puis v_6 avant de revenir à v_7 puis à v_5 pour s'arrêter. Les instants de fin de traitement des sommets lors du parcours récursif sont donc dans l'ordre suivant :

$$t_0 < t_4 < t_2 < t_3 < t_1 < t_6 < t_7 < t_5. \tag{2}$$

La liste renvoyée par `dfs_tri` est donc `[5; 7; 6; 1; 3; 2; 4; 0]`.

b) Le parcours en profondeur du graphe renversé G', en considérant les sommets de départ dans l'ordre inverse de (2), choisit d'abord v_5 et visite ainsi v_6 et v_7, ce qui donne la composante $\{v_5, v_6, v_7\}$. Ensuite le parcours reprend du sommet v_1 et visite ainsi v_4, ce qui donne la composante $\{v_1, v_4\}$. Ensuite le parcours reprend du sommet v_3 et visite ainsi v_2, ce qui donne la composante $\{v_2, v_3\}$. Enfin le parcours reprend du sommet v_0 et s'arrête là, ce qui donne la composante $\{v_0\}$.

Codage de l'algorithme. L'étape a) de l'algorithme est déjà codée dans `dfs_tri`.

Question 4 Justifier formellement la complexité linéaire (en la taille du graphe) de la fonction `dfs_tri`. Rappelons que la taille du graphe est la somme de son nombre de sommets et de son nombre d'arêtes.

Pour pouvoir effectuer l'étape b) il faut d'abord renverser le graphe.

Question 5 Écrire une fonction `renverser_graphe` qui prend en entrée un graphe et qui renvoie un autre graphe dans lequel les sommets sont les mêmes et le sens de toutes les arêtes est inversé. La complexité de la fonction doit être linéaire en la taille du graphe fourni en entrée.

```
renverser_graphe: graphe -> graphe
```

Question 6 Écrire une fonction `dfs_cfc` qui code l'étape b) de l'algorithme. Elle prend en entrée le graphe renversé construit à la question précédente, ainsi que la liste d'entiers renvoyée par `dfs_tri` sur le graphe de départ. Elle renvoie une liste dans laquelle chaque élément est une liste d'entiers contenant les indices des sommets d'une composante fortement connexe du graphe, sans doublons. La complexité de la fonction doit être linéaire en la taille du graphe.

```
dfs_cfc: graphe -> int list -> int list list
```

Question 7 Écrire enfin une fonction `cfc` 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 fortement connexe étant stockée dans l'une des listes d'entiers. La complexité de la fonction doit être linéaire en la taille du graphe.

```
cfc: graphe -> int list list
```

Correction de l'algorithme. Nous dirons qu'une composante fortement connexe C de G est *subordonnée* à une autre composante fortement connexe C' s'il existe des sommets $v_i \in C$ et $v_{i'} \in C'$ et un chemin de $v_{i'}$ à v_i dans G. Comme C et C' sont des composantes fortement connexes, cela revient à dire qu'il existe des chemins dans G de n'importe quel sommet de C' à n'importe quel sommet de C.

Question 8 Montrer que la relation *être subordonnée à* est une relation d'ordre (pas forcément totale) sur l'ensemble des composantes fortement connexes de C.

À chaque composante fortement connexe C on associe un instant t_C de fin de traitement comme ceci :

$$t_C = \max_{v_i \in C} t_i,$$

où les t_i sont définis comme à l'étape a) de l'algorithme. Par exemple, dans l'exemple de la figure 2 on a : $t_0 = t_{\{v_0\}} < t_3 = t_{\{v_2,v_3\}} < t_1 = t_{\{v_1,v_4\}} < t_5 = t_{\{v_5,v_6,v_7\}}$.

Question 9 Montrer que pour tous sommets $v_i \neq v_j$ tels qu'il existe un chemin de v_i à v_j dans le graphe et pas de chemin de v_j à v_i, on a $t_i > t_j$.

Question 10 Montrer que l'ordre total sur les composantes fortement connexes de G défini par les instants de fin de traitement t_C est compatible avec l'ordre partiel *être subordonnée à*, c'est-à-dire que pour toutes composantes C et C' telles que C est subordonnée à C', on a $t_C \leq t_{C'}$.

Question 11 En utilisant les résultats des questions précédentes, montrer que le parcours en profondeur du graphe renversé G' à l'étape b) de l'algorithme extrait les composantes fortement connexes de G les unes après les autres, dans l'ordre des t_C décroissants. Pour cela on pourra s'appuyer sur les observations simples suivantes sans les démontrer :

- Les composantes fortement connexes de G' sont les mêmes que celles de G.
- La relation d'ordre *être subordonnée à* dans G' est inversée par rapport à celle dans G.

II.2 Des composantes fortement connexes à 2-SAT

La réduction d'une instance de 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 $(\neg l_j) \Rightarrow l_i$. 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 $x_0, \cdots, x_{r-1}$) :

- 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 $\neg x_i$.
- Pour chaque clause de type (l_i) 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 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 de 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 $(\neg x_i \vee x_i)$, soit on se ramène au cas d'une clause de type (l_i) si elle est de la forme $(x_i \vee x_i)$ ou $(\neg x_i \vee \neg x_i)$.

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.

Question 12 Donner le résultat de la procédure ci-dessus sur la formule $(x_1 \vee x_2) \wedge x_0 \wedge (x_2 \vee \neg x_2) \wedge (\neg x_2 \vee x_0)$.

Question 13 Écrire une fonction `deux_sat_vers_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

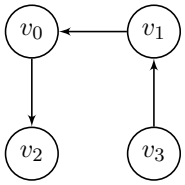

FIGURE 3 – Le graphe obtenu à partir de la formule $x_0 \wedge (x_1 \vee \neg x_0)$.

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 `deux_sat_vers_graphe` doit être linéaire en la taille de f.

```
deux_sat_vers_graphe: fnc -> graphe
```

Question 14 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 la notation $\lfloor \cdot \rfloor$ désigne la partie entière inférieure.

Question 15 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} se trouvent 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} se trouvent 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 σ telle que $\sigma(f) =$ `Vrai`. Ceci fournit un critère simple pour décider de la satisfiabilité de f.

Question 16 Écrire une fonction `deux_sat` 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.

```
deux_sat: fnc -> bool
```

Partie III. 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 `Vrai`, 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 I. Ce type nous fait travailler non plus dans l'algèbre binaire 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 dans la figure 4.

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

$a \vee b$		b		
		Vrai	Indét.	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. Les valeurs d'affectation des variables sont respectivement Vrai, Indét(erminé), Faux.

Question 17 Écrire trois fonctions : `et`, `ou`, `non`, qui codent respectivement les connecteurs logiques $\wedge$, $\vee$, $\neg$ sur les triléens. La complexité de chaque fonction doit être constante.

```
et: trileen -> trileen -> trileen
ou: trileen -> trileen -> trileen
non: trileen -> trileen
```

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.

Question 18 En vous appuyant sur la remarque ci-dessus et sur les fonctions de la question 17, écrire une fonction `eval` 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.

```
eval: fnc -> trileen vect -> trileen
```

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.

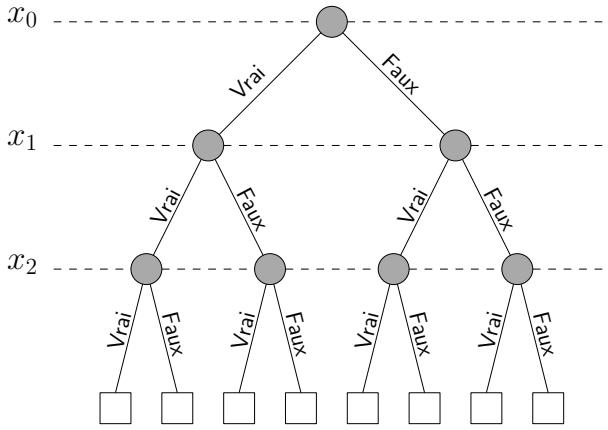

FIGURE 5 – Arbre parcouru lors de la recherche exhaustive parmi les valuations des variables x_0, x_1, x_2.

Au départ la valeur `Indetermine` est affectée à toutes les variables. Le parcours commence à la racine. À chaque nœud 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 nœud 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`.

Question 19 Écrire une fonction `k_sat` 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. Sa complexité doit être en $\mathcal{O}(n\,2^n)$, où n est la taille de f. En outre, un soin particulier doit être apporté à la clarté du code, dans lequel il est recommandé d'insérer des commentaires aux points clés.

```
k_sat: fnc -> bool
```

Partie IV. De k-SAT à SAT

Dès le début du sujet nous avons laissé de côté 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 $k \geq 3$) par un algorithme de complexité polynomiale. Ainsi, l'algorithme codé à la question 19, 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.

Question 20 Pour chacune des formules suivantes, utiliser l'involutivité de la négation, 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 20 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 sera polynomiale en n. La procédure pour construire f' à partir de f fonctionne 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 nœuds 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 20, 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.

Question 21 Montrer que les formules f et f' sont équisatisfiables.

Question 22 Écrire une fonction `negs_en_bas` 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 `negs_en_bas` doit avoir une complexité linéaire en la taille de f.

```
negs_en_bas: formule -> formule
```

On se donne à présent une nouvelle fonction `var_max`, 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.

Question 23 Écrire une fonction `formule_vers_fnc` qui prend en argument la formule f^* obtenue à l'issue de l'étape 1 et qui renvoie la FNC f' construite comme à l'étape 2. La complexité de la fonction `formule_vers_fnc` doit être polynomiale en la taille de f^*.

```
formule_vers_fnc: formule -> fnc
```

Question 24 Justifier les complexités des fonctions `negs_en_bas` et `formule_vers_fnc`. On pourra par exemple montrer que le nombre de clauses formées dans `formule_vers_fnc` est égal au nombre de littéraux dans la formule f^*.

Ainsi, il suffit de combiner les fonctions des questions 22 et 23 pour convertir n'importe quelle formule booléenne f en une FNC f' équisatisfiable en temps polynomial.

*
* *

De manière similaire, on peut convertir la FNC f' en une 3-FNC f'' équisatisfiable en temps polynomial. L'approche est la suivante : toute clause de f' avec au plus 3 littéraux reste inchangée, tandis que toute clause $\bigvee_{j=1}^{n_i} l_{ij}$ avec $n_i > 3$ devient :

$$(l_{i1} \vee l_{i2} \vee x_{i1}) \wedge (\neg x_{i1} \vee l_{i3} \vee x_{i2}) \wedge \cdots \wedge (\neg x_{i(n_i-4)} \vee l_{i(n_i-2)} \vee x_{i(n_i-3)}) \wedge (\neg x_{i(n_i-3)} \vee l_{i(n_i-1)} \vee l_{in_i}),$$

où $x_{i1}, \cdots, x_{i(n_i-3)}$ sont de nouvelles variables introduites spécialement pour cette clause. On peut alors montrer que la formule f'' ainsi obtenue est de taille polynomiale en la taille de f' et équisatifsiable. Dès lors, l'instance initiale de SAT est transformée en une instance de 3-SAT (et donc de k-SAT pour n'importe quel $k \geq 3$) équisatisfiable et de taille polynomiale.

La méthode décrite dans la partie IV du sujet transforme toute instance de SAT en une instance de 3-SAT équisatisfiable en temps polynomial. Dès lors, la fonction `eval` de la question 18 permet, étant données une formule booléenne f quelconque de taille n à r variables $x_0, \cdots, x_{r-1}$ et une valuation $\sigma : \{x_0, \cdots, x_{r-1}\} \to \{\texttt{Vrai}, \texttt{Faux}\}$, de vérifier en temps polynomial (en n, rappelons que l'on a $r \leq n$) si $\sigma(f) = \texttt{Vrai}$. Ainsi, du point de vue de la théorie de la complexité, le problème SAT se trouve être dans la classe de complexité NP. Cette classe est constituée précisément des problèmes dont on peut vérifier la solution en temps polynomial quand cette dernière existe. Cela ne veut pas dire qu'une telle solution peut être trouvée (ni même son existence déterminée) en temps polynomial. Les problèmes pour lesquels c'est le cas forment la classe de complexité appelée P. Clairement, on a P⊆NP puisque si l'on peut trouver une solution en temps polynomial alors on peut a fortiori vérifier en temps polynomial qu'on a bien une solution. Savoir si P est égal à NP est une question encore ouverte à ce jour et sans conteste l'une des plus fameuses en informatique fondamentale.

Parmi les problèmes de la classe NP, certains sont particulièrement difficiles, au moins aussi difficiles que tous les autres. Ces problèmes sont dits *NP-complets*. Formellement, un problème $P_1 \in$ NP est dit NP-complet s'il existe une *réduction polynomiale* de chaque problème $P_2 \in$ NP vers ce problème, c'est-à-dire un algorithme qui transforme toute instance u de P_2 en une instance v de P_1 en temps polynomial, de telle sorte que la réponse ("oui" / "non") de P_1 sur l'instance transformée v est la même que celle de P_2 sur l'instance initiale u. Un problème NP-complet permet ainsi de résoudre les instances de n'importe quel autre problème de la classe NP. D'où l'importance des problèmes NP-complets, puisque si l'un d'eux s'avérait

être dans P alors tous les autres y seraient également et l'on aurait P=NP. A contrario, si l'on avait P≠NP alors aucun problème NP-complet ne serait résoluble en temps polynomial.

Historiquement, SAT fut le premier problème à être identifié comme étant NP-complet. C'est le fameux théorème de Cook-Levin (1971). Ce résultat explique l'importance qu'a eue SAT dans le développement de la théorie de la complexité. La transformation décrite dans la partie IV du sujet donne une réduction polynomiale de SAT à 3-SAT. Dès lors, 3-SAT (ainsi que k-SAT pour tout $k \geq 3$) est également NP-complet. Par conséquent, l'algorithme exponentiel de la question 19 permet de résoudre toutes les instances de tous les problèmes de la classe NP. A contrario, si P≠NP alors il n'existe pas d'algorithme permettant de résoudre chacune des instances de 3-SAT (ou de k-SAT pour $k \geq 3$) en temps polynomial. On comprend dès lors l'ampleur du fossé qui sépare les petites valeurs de k ($k = 1, 2$), pour lesquelles des algorithmes linéaires existent comme décrit dans les parties I et II du sujet, des valeurs plus grandes ($k \geq 3$), pour lesquelles il n'existe peut-être pas d'algorithme polynomial. Ceci est l'un des nombreux effets de seuil observés dans la complexité du problème SAT et de ses variantes.

*
* *