Satisfiabilité

1 Introduction

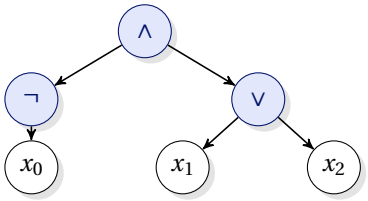
1.1 Formules propositionnelles

On rappelle qu'une *variable propositionnelle* est une variable prenant ses valeurs dans l'ensemble des booléens $\mathcal{B} = \{\mathbf{V}, \mathbf{F}\}$. Une *formule propositionnelle* s'obtient en combinant des variables propositionnelles avec les opérateurs logiques binaires de conjonction ($\wedge$), de disjonction ($\vee$) et l'opérateur logique unaire de négation ($\neg$).

On implémente les formules propositionnelles en OCaml avec le type suivant :

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

Ainsi les formules propositionnelles sont représentées par des structures arborescentes en machine, appelées *arbres d'expression* dans la suite. Par exemple, l'arbre ci-dessous est associé à la formule $(\neg x_0) \wedge (x_1 \vee x_2)$.



Cette même formule s'écrira en OCaml :

```
Et (Non (Var 0), Ou (Var 1, Var 2))
```

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 $x_0, \ldots, x_{r-1}$.

On considérera $\neg$ prioritaire sur $\wedge$ et $\vee$, et $\wedge$ prioritaire sur $\vee$, de façon à limiter l'usage des parenthèses. Ainsi, $(\neg x_0) \wedge (x_1 \vee x_2)$ pourra s'écrire plus simplement $\neg x_0 \wedge (x_1 \vee x_2)$.

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 (soit 6 dans l'exemple précédent), et on a naturellement $r \leqslant n$.

1.2 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 distribution de vérité μ telle que l'évaluation de f pour cette distribution μ vérifie $\varepsilon_\mu(f) = \mathbf{V}$. 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.

1. Pour chaque formule ci-dessous, 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) $(x_0 \vee x_1) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_1)$.

Dans la suite, on s'intéresse à des formules écrites sous forme normale conjonctive (FNC) :

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

où chaque *littéral* $l_{i,j}$ est soit une variable propositionnelle x_k, soit sa négation $\neg x_k$, les littéraux étant regroupés en *clauses disjonctives* $\bigvee_{j=1}^{n_i} l_{i,j}$. Par exemple, les formules a), b) et d) de la question précédente 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 $i \in [\![1 \,..\, m]\!]$. Notez que la formule est également une $k'-$FNC pour tout $k' > k$. Les formules a), b) et d) de la question précédente sont des 2-FNC.

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, la formule a) précédente sera représentée par

```
[ [ V 1 ]; [ V 0; NV 0 ]; [ NV 1 ] ]
```

2. Proposer une fonction `dim` de signature `fnc -> int` prenant en argument une FNC f et renvoyant le plus petit k tel que f soit une $k-$FNC. La complexité de la fonction doit être linéaire en la taille de f.

3. 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.

4. Écrire une fonction `taille` qui prend en entrée une FNC f et renvoie sa taille. La complexité de la fonction doit être linéaire en la taille de f.

2 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 **V**, soit $l_i = \neg x_i$ et dans ce cas x_i doit valoir **F**. 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` suivant :

```
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.

5. Écrire une fonction `un_sat` de signature `fnc -> bool array option` qui prend en entrée une FNC f, supposée être une 1−FNC, et qui renvoie `None` si f est insatisfiable, et `Some tab` où `tab` est un tableau de booléens représentant un modèle μ pour f si f est satistiable (dans la case d'index i de `tab`, on trouvera $\mu(x_i)$). La complexité de la fonction doit être linéaire en la taille de f.

Remarque : le type `trileen` permet d'analyser la formule, mais on souhaite bien un tableau de *booléens* pour le modèle retourné.

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 distributions de vérité possibles. Pour chaque distribution de vérité considérée on évalue la formule : si le résultat est **V** alors on a trouvé un modèle, si le résultat est **F** alors on rejette la distribution de vérité 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 distributions de vérité partielles et décide, soit d'accepter la distribution de vérité partielle courante si le résultat de l'évaluation est déjà **V**, soit de la rejeter précocement si le résultat est déjà **F**, soit enfin de compléter la construction de la distribution de vérité si le résultat de l'évaluation est encore indéterminé.

Pour coder les évaluations à partir d'une distribution de vérité partielle en machine, nous allons utiliser des tableaux de triléens. 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 $\wedge$ et $\neg$ sont données ci-dessous :

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

a	$\neg a$
Vrai	Faux
Indet.	Indet.
Faux	Vrai

6. Donner la table en logique de Kleene pour $\vee$.

7. Écrire une fonction `et` de signature `trileen -> trileen -> trileen` qui code le connecteur logique $\wedge$. La complexité doit être constante.

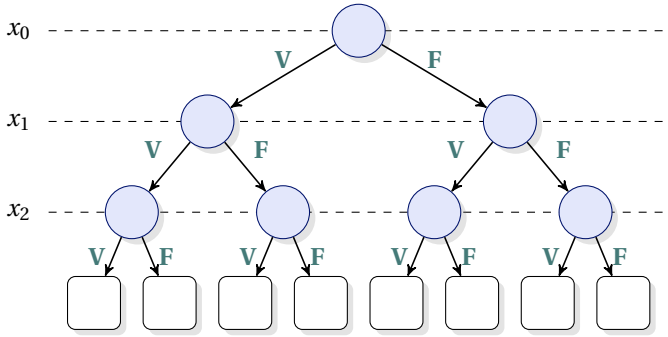
On suppose dans la suite avoir codé deux fonctions similaires, `ou` et `non`, codant $\vee$ et $\neg$.

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.

8. En vous appuyant sur la remarque ci-dessus et sur les fonctions de la question précédente, écrire une fonction `eval` de signature `fnc -> trileen vect -> 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 distribution de vérité 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. On pourra commencer par proposer une fonction `eval_clause` faisant la même chose sur une clause, puis utiliser

cette dernière pour le cas d'une FNC.

Nous pouvons maintenant décrire l'algorithme de recherche exhaustive avec terminaison précoce. Pour itérer sur l'ensemble des valuations nous utilisons une approche s'inspirant du principe du retour sur trace, en parcourant 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 ci-dessous pour le cas de 3 variables.



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 distribution de vérité partielle courante on utilise un tableau de triléens dans lequel les variables non encore affectées prennent la valeur `Indetermine`.

9. Écrire une fonction `k_sat` de signature `fnc -> bool array option` qui prend en entrée une FNC `f` et qui renvoie `None` si `f` n'est pas satisfiable et `Some tab` où `tab` est un tableau de booléens représentant un modèle sinon (attention, on attend bien un tableau de booléens et non de triléens !). La fonction doit coder la méthode de recherche exhaustive avec terminaison précoce décrite ci-dessus. Sa complexité doit être en $O(n2^n)$ dans le pire des cas, 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.

4 Problème 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 $k \geqslant 3$) par un algorithme de complexité polynomiale. Ainsi, l'algorithme codé à la question précédente, 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.

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

Le second exemple de la question précédente 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 précédente, 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 $f^* = \varphi^* \wedge \psi^*$, alors on pose $f' = \varphi' \wedge \psi'$, où φ' et ψ' sont les versions réécrites de φ^* et ψ^* respectivement,
   — si $f^* = \varphi^* \vee \psi^*$, alors on introduit une nouvelle variable booléenne x dans la formule et on pose $f' = \bigwedge_{i=1}^{p}(\varphi'_i \vee x) \wedge \bigwedge_{j=1}^{q}\left(\psi'_j \vee \neg x\right)$ où $\varphi' = \bigwedge_{i=1}^{p} \varphi_i$ 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.

11. Montrer que les formules f et f' sont équisatisfiables.

12. Écrire une fonction `negs_en_bas` de signature `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 `negs_en_bas` doit avoir une complexité linéaire en la taille de f.

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.

13. Écrire une fonction `formule_vers_fnc` de signature `formule -> 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^*.

14. 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 squestions 12 et 13 pour convertir n'importe quelle formule booléenne f en une FNC f' équisatisfiable en temps polynomial.

15. Si la démarche précédente détermine si la fonction est satisfiable, elle ne fournit pas complètement un modèle. Expliquer comment il serait possible d'obtenir un modèle à partir de la distribution triléenne construit par la démarche de retour sur trace de la section précédente.