

TP5. Le problème 2-SAT en OCaml

On étudie dans ce TP le problème 2-SAT :

2-SAT : $\left\{ \begin{array}{l} \text{Entrée : } m \text{ clauses } C_1, \dots, C_m \text{ de taille 2 portant sur des variables } (x_i)_{1 \leq i \leq n} \\ \text{Sortie : } \text{La formule } \varphi = C_1 \wedge \dots \wedge C_m \text{ est elle satisfiable ?} \end{array} \right.$



La formule:

$f_2 = (x_3 \vee x_2) \wedge (x_1 \vee x_2) \wedge (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_4) \wedge (x_1 \vee \neg x_1) \wedge (x_4 \vee \neg x_3) \\ \wedge (\neg x_4 \vee \neg x_1) \wedge (\neg x_3 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_4)$ est elle satisfiable ?

Si oui, en donner un modèle.

Les **formules** logiques seront représentées à l'aide des types et valeurs suivants :

```
type clause = int * int
type formule = clause list
exception Fausse
type valuation = int array
type sommets = int list
```

Dans une **formule**, on notera n le nombre de variables, qui seront indicées de 1 à n : l'entier i représentera le **littéral** x_i , $-i$ le littéral $\neg x_i$. On note m le nombre de clauses.

On dispose dans TP5.ml de trois formules :

- f_1 et f_0 respectivement **satisfiable** et **non satisfiable**, ainsi qu'un **modèle** m_1 pour f_1 .
- Le but du TP est de savoir si f_2 est satisfiable.

Une **valuation** sera représenté par un tableau v de taille $n + 1$, tel que $v.(i)$ vaille $+i$ si x_i est évaluée à **vrai** et $-i$ si x_i est évaluée à **faux**. $v.(0)$ ne représente rien. Cette représentation d'une valuation (on aurait aussi pu prendre un tableau de booléen) est plus facile à lire pour un humain.

1. Écrire `check_clause : valuation -> clause -> bool` qui teste la satisfiabilité d'une clause.
2. Écrire une fonction `interpretation : formule -> valuation -> bool` qui renvoie la valeur d'une formule par une valuation. On utilisera `List.for_all`.
3. Vérifier que `m1` est un modèle pour `f1`.
4. Donner la complexité asymptotique de `interprétation` en fonction de n et/ou m .

I - Rappel : l'algorithme de Quine

La formule vide (sans clause) est vraie par hypothèse.

On note $\varphi^{\{x_i \leftarrow a\}}$ la formule obtenu à partir d'une formule φ non vide, dans laquelle on a substitué $a \in \{V, F\}$ à x_i dans φ .

Pour simplifier une formule non vide, on procède ainsi (on envisage les cas dans cet ordre):

- Si une clause est satisfaite par la substitution, on la supprime de la formule.
- Si une clause est insatisfiable par la substitution, la formule devient $\perp$
- Si une clause est de la forme (l_i, l_j) ou (l_j, l_i) telle que l_i soit faux, on la remplace par (l_j, l_j) (c'est plus simple informatiquement).
- Sinon, on laisse la clause inchangée.

5. Donner $f_2^{\{x_1 \leftarrow V\}}$ et $f_2^{\{x_1 \leftarrow F\}}$.

6. Écrire la fonction `simplification : formule -> int -> formule` qui renvoie la formule associée à la substitution $\{x_i \leftarrow V\}$ si $i > 0$, et $\{x_{-i} \leftarrow F\}$ si $i < 0$.

On déclenche l'exception `Fausse` (instruction `raise Fausse`) si on rencontre une clause fausse : en effet il ne sert à rien de continuer à simplifier.

7. La tester sur f_2 pour diverses substitutions.

On rappelle l'algorithme de QUINE :

Algorithme 1 : Algorithme de QUINE.

Données :

- f : formule
- i : indice de variable courant

Résultat : booléen : `true` si f satisfiable, `false` sinon.

```

1 Fonction Quine( $f, i$ ):
2   si  $f = \top$  alors retourner true
3   si Quine( $f^{\{x_i \leftarrow V\}}, i + 1$ ) alors retourner true
4   si Quine( $f^{\{x_i \leftarrow F\}}, i + 1$ ) alors retourner true
5   retourner false

```

8. Écrire une fonction `quine : formule -> int -> bool`. Pour les formules fausses on rattrapera l'exception `Fausse` par `try ... with |Fausse -> false`. On appelle la fonction avec $i = 1$.

9. Vérifier la validité de `quine` sur f_0 et f_1 .

10. Que peut-on dire sur f_2 ?

II - Une méthode polynomiale pour résoudre 2-SAT

On transforme le problème de la satisfiabilité en un autre, celui de la recherche de composantes connexes dans un graphe. La justification sera donnée en cours.

Pour se faire on associe à toute formule φ de 2-SAT son graphe d'implication $G_\varphi = (S, A)$ orienté défini comme suit :

- $S = \{x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\}$
- $A = \{(\neg l_i, l_j) | l_i \vee l_j \in \varphi\} \cup \{(\neg l_j, l_i) | l_i \vee l_j \in \varphi\}$

Le nom du graphe vient de la propriété $l_i \vee l_j \equiv \neg l_i \rightarrow l_j \equiv \neg l_j \rightarrow l_i$. Chaque arc est associé à une implication logique. On montre(ra) que la formule est satisfiable si et seulement si $\forall i, x_i$ et $\neg x_i$ ne sont pas dans la même composante fortement connexe.

11. Dessiner le graphe d'implication G_2 associé à f_2 . Identifier les composantes fortement connexes. La formule f_2 est elle satisfiable ?



Un sommet sera représenté par un entier `type sommet = int`.

On rappelle que précédemment, on a associé l'entier positif i au littéral x_i , et l'entier négatif $-i$ au littéral $\neg x_i$.

Comme les tableaux n'acceptent pas les indices négatifs, on décale la numérotation:

Au littéral représenté par $i \in \llbracket -n \dots n \rrbracket$ on associe le sommet $s_i = n + i \in \llbracket 0 \dots 2n \rrbracket$. Le sommet n représente un sommet isolé qui ne gêne pas dans la suite. Ainsi une clause (i, j) sera associée aux arcs $(n - i, n + j)$ et $(n - j, n + i)$.

Un graphe d'implications à $2n$ sommets, associé à une formule à n variables, sera donc représenté par un tableau de liste d'adjacence de taille $2n + 1$, de type `type graphe = sommet list array`.

12. Numéroté les $2n + 1$ sommets sur la figure précédente.

13. Écrire une fonction `graphe_implication : formule -> int -> graphe` telle que `graphe_implication f n` génère le graphe associé à la formule f , où n correspond au nombre de variables. On définira une fonction locale `remplir : clause -> unit` que l'on itérera sur `f` avec `List.iter`. Enfin on utilisera `Array.map` et la fonction `tri_sans_doublons` fournie pour obtenir des listes d'adjacences triées sans doublons. Définir `g2 = graphe_implication f2 4` et vérifier.

14. Écrire une fonction `graphe_T : formule -> int -> graphe` qui construit le graphe transposé. On remarquera qu'il suffit de transformer `f` par la transformation $(i, j) \rightarrow (-i, -j)$ avec `List.map`, et d'utiliser `graphe_implication`.

15. Écrire une fonction `tri_topologique : graphe -> sommet list` qui renvoie l'ordre postfixe d'un parcours en profondeur d'un graphe d'implication (qui correspond donc à un tri topologique du graphe).

Attention : On rappelle que dans un graphe d'implication les indices 0 et n du tableau ne correspondent pas à des sommets.

16. En déduire une fonction `kosaraju : graphe -> int array` qui renvoie les composantes fortement connexes d'un graphe d'implication.

17. Finalement, écrire une fonction `satisfiable : formule -> bool` qui résout 2-SAT en vérifiant que pour toute variable x_i , x_i et $\neg x_i$ ne sont pas dans la même composante fortement connexe du graphe d'implication.