TP 2

# Résolution polynomiale de 2-SAT

MPI/MPI*, lycée Faidherbe

**Résumé**

On utilise dans ce TP des formules logiques sous forme normale conjonctive avec au plus 2 littéraux par clause.

Après avoir écrit un vérificateur polynomial, on mettra en œuvre un algorithme polynomial de résolution d'une instance de 2-SAT en utilisant l'algorithme de Kosaraju.

## Présentation

On se base sur le fichier `TP2.ml`.
Les **formules** logiques seront représentées à l'aide des alias suivants :

```
type clause = int * int
type formule = clause list
type valuation = int array
type sommets = int list
type graphe = sommets array
```

Dans une formule, on notera $n_v$ 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 $n_c$ le nombre de clauses.
Une valuation sera représenté par un tableau `v` de taille $n_v + 1$ : `v.(i)` vaut i si $x_i$ est valuée à **vrai** et $-i$ si $x_i$ est valuée à **faux** . `v.(0)` ne représente rien.
la fonction `formule : int -> int -> int -> formule` est fournie.
`formule nv nc seed` génère une formule aléatoire à $n_v$ variables, $n_c$ clauses et deux littéraux par clause. Les deux littéraux d'une clause peuvent être égaux, et deux clauses d'une formule peuvent être équivalentes. `seed` est un germe qui initialise le générateur aléatoire : la même valeur de `seed` génèretoujours la même formule.
Des exemples sont donnés, construits avec la fonction `formule` :
On dispose dans `TP2.ml` de quatre formules tests :

- $f_0$ n'est pas satisfiable,
- $f_1$ est satisfiable,
- $m_1$ est un modèle pour $f_1$,
- le but du TP est de savoir si $f_2$ est satisfiable.
- $f_3$ est une formule simple.

## I Vérification

**Question 1** Écrire `check_clause (v:valuation) (c:clause) : bool` qui teste la satisfiabilité d'une clause $C$ var la valuation $v$.

```
let check (v : valuation) (c : clause) =
  let l1, l2 = c in
  v.(abs l1) = l1 || v.(abs l2) = l2
```

**Question 2** Écrire une fonction `interpretation (f:formule) (v:valuation) : bool` qui renvoie $v(f)$. On pourra utiliser `List.for_all`.

```
let interpretation (f : formule) (v ; valuation) =
   List.for_all (check v) f
```

**Question 3** Vérifier que `m1` est un modèle pour `f1`

```
interpretation f1 m1
```

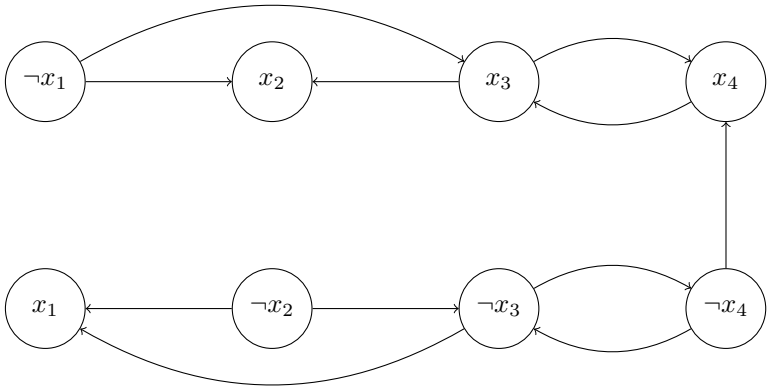
# II Graphe associé

On associe à toute formule $\varphi$ de 2-SAT son **graphe d'implication** défini comme suit :

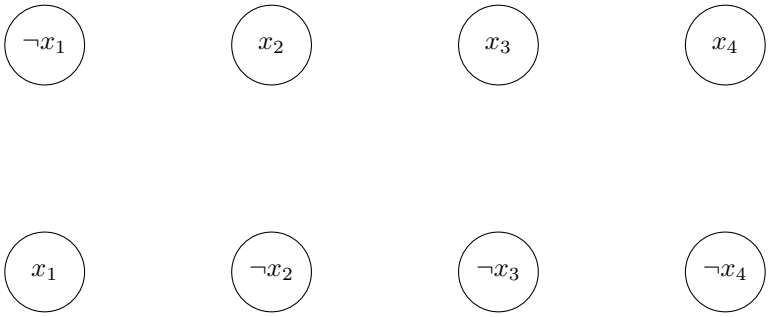
- $S = \{x_1, \dots, x_n, \dots, \neg x_1, \dots, \neg x_n\}$
- $A = \big\{(\neg \ell_i, \ell_j), (\ell_i \vee \ell_j) \in \varphi\big\} \cup \big\{(\neg \ell_j, \ell_i), (\ell_i \vee \ell_j) \in \varphi\big\}$.

Le nom du graphe vient de la propriété $\ell_i \vee \ell_j \equiv \neg \ell_i \to \ell_j \equiv \neg \ell_j \to \ell_i$ et qu'on peut associer à un arc $(a, b)$ l'implication logique $a \to b$.

**Question 4** Dessiner le graphe d'implication $G_3$ associé à $f_3$.
Identifier les composantes fortement connexes.
L'inversion de $x_1$ avec $\neg x_1$ est volontaire.




*Les seules composantes fortement connexes non réduites à un point sont* $\{x_3, x_4\}$ *et* $\{\neg x_3, \neg x_4\}$




Comme un littéral peut être négatif, on décale la numérotation pour le tableau des listes d'adjacences du graphe associé à une 2-clause. Au littéral représenté par $i$ ($-n \leqslant l \leqslant n$) on associe le sommet $s_i = n + i$, $0 \leqslant s_i \leqslant 2n$. Le sommet $n$ représente un sommet isolé qui ne gène pas dans la suite. Ainsi une clause $(i, j)$ sera associée aux arcs $(n - i, n + j)$ et $(n - j, n + i)$.

L'algorithme de KOSARAJU (simplifié), est décrit ci-dessous :

**Algorithme 1 :** Algorithme de KOSARAJU simplifié

**def** `kosaraju_2sat(`$f$,$n$`)` $=$
  $G \leftarrow$ `graphe(`$f$,$n$`)`
  $G^t \leftarrow$ `graphe_T(`$f$,$n$`)`
  $opi_1 \leftarrow$ `opi(`$G^t$, $S$`)`
  $opi_2 \leftarrow$ `opi(`$G$, $opi_1$`)`
  **retourner** $opi_2$

`opi(G, l)` renvoie l'ordre postfixe inverse pour un graphe $G$ avec une lecture des sommets dans l'ordre donné par la liste `l`.

**Question 5** Définir une fonction `graphe (f:formule) (n:int) : graphe` qui génère le graphe associé à la formule $f$ ; $n$ représente le nombre de variables.
On renverra des listes de voisins **sans doublon**.

---

*On peut enlever les doublons en temps linéaire.*

```
let sans_doublon l n =
  let h = Hashtbl.create n in
  let rec ajouter reste fait  =
    match reste with
    |[] -> fait
    |t :: q -> if Hashtbl.mem h t
               then ajouter q fa it
               else begin
                 Hashtbl.add h t 1;
                 ajouter q (t :: fait)
               end in
  ajouter l []
```

```
let graphe (f:formule) (n:int) =
  let g = Array.make (2*n+1) [] in
  let plusArc (i,j) =  g.(n+i)<- n+j::g.(n+i) in
  let remplir (i, j) =
    plusArc (-i, j);
    plusArc (-j, i) in
  List.iter remplir f;
  for i = 0 to (2*n) do
    g.(i) <- sans_doublon  g.(i) n done;
  g
```

**Question 6** Définir une fonction `graphe (f:formule) (n:int) : graphe` qui renvoie le graphe transposé du graphe associé à la formule $f$

---

*On peut calculer directement le graphe transposé*

```
let grapheT (f:formule) (n:int) =
  let g = Array.make (2*n+1) [] in
  let plusArc (i,j) =  g.(n+i)<- n+j::g.(n+i) in
  let remplir (i, j) =
    plusArc (i, -j);
    plusArc (j, -i) in
  List.iter remplir f;
  for i = 0 to (2*n) do
    g.(i) <- sans_doublon  g.(i) n done;
  g
```

*On peut aussi calculer une fonction de transposition de graphe*

```
let transposer (g : graphe) : graphe =
 let n = Array.length g in
  let gt = Array.make n [] in
  for i = 0 to (n-1) do
    List.iter (fun j -> gt.(j) <- (i :: gt.(j))) g.(i) done;
  gt
```

*Alors on peut écrire*

```
let grapheT (f:formule) (n:int) =
  transposer (graphe f n)
```

---

**Question 7** Définir une fonction `opi (g:graphe) (s:sommets) : sommets` qui renvoie la liste des sommets explorés par un parcours en profondeur de `g`, dans l'ordre postfixe inversé et en lisant les sommets dans l'ordre donné par `s`.

```
let opi (g : graphe) (s : sommets) : sommets=
  let n = Array.length g in
  let vu = Array.make m false  in
  let out  = ref [] in
  let explorer s =
    if  not vu.(s)
    then begin
      vu.(s) <-true;
      List.iter explorer g.(s);
      out := s :: !out
    end in
  List.iter explorer s;
  !out
```

Pour tester on peut utiliser `List.init n abs` qui renvoie une liste conteant les entier de 0 à $n-1$.

**Question 8** Écrire `kosaraju_2sat f n` correspondant à l'algorithme fourni.

```
let kosaraju_2sat f n : sommets  =
  let g = graphe f n in
  let gt = grapheT f n in
  let s0 = List.init (2*n+1) abs in
  let s1 = opi gt s0 in
  opi g s1
```

Dans le calcul de la valuation validant une forme 2-sat, on regarde, pour chaque variable $x_i$, quel est le littéral placé avant l'autre dans l'ordre topologique des composantes fortement connexe. L'ordre postfixe inverse est un ordre topologique des CFC donc, si $\ell$ est placé avant $\overline{\ell}$, $\ell$ doit prendre la valeur `false`.

**Question 9** Prouver que si on donne la valeur **vrai** aux littéraux dans l'ordre fourni par l'algorithme alors, dans le cas ou la formule est satisfiable, cela donne un modèle.

*Dans le cours, l'ordre postixe direct est utilisé pour reconnaître les CFC, elles apparaissent en entier dans un ordre topologique inversé. L'ordre postfixe inverse est donc un ordre topologique des CFC. Si $\ell$ est placé avant $\overline{\ell}$ dans cet ordre on donne la valeur **faux** à $\ell$, c'est bien ce que fait l'algorithme proposé qui donne la valeur **vrai** à $\ell$ puis la valeur **vrai** à $\overline{\ell}$ donc bien la valeur **faux** à $\ell$.*

**Question 10** En déduire une fonction `calculVal (f : formule) (n : int) : valuation` qui calcule cette valuation.

```
let calculVal f n =
  let s = kosaraju_2sat f n in
  let valu = Array.make (n+1) 0 in
  List.iter (fun i -> valu.(abs(i-n)) <- i-n) s;
  valu
```

**Question 11** En déduire une fonction `satisfiable f n : valuation` qui renvoie un modèle quand `f` est satisfiable, et déclenche une exception sinon.

```
exception NonSatisfiable

let satisfiable f n =
  let v =calculVal f n in
  if interpretation f v
  then v
  else raise NonSatisfiable
```

**Question 12** Écrire le vrai `kosaraju (g:graphe) : sommets list` qui renvoie les composantes fortement connexes d'un graphe.

```
let kosaraju_vrai g   =
  let n = Array.length g in
  let vus = Array.make n false in
  let cfc = ref [] in
  (* création de la liste des sommets dans le bon ordre *)
  let gt = transposer g in
  let s0 = List.init n abs in
  let s1 = opi gt s0 in
  (* visite des éléments *)
  let rec visiter i =
    if not vus.(i)
    then begin
      vus.(i) <- true;
      cfc := i :: !cfc;
      List.iter visiter g.(i)
    end in
  (* recherche des CFC *)
  let rec traiter a_voir liste_CFC =
    match a_voir with
    |[] -> liste_CFC
    |k :: reste ->
        if vus.(k)
        then traiter reste liste_CFC
        else begin
          cfc := [];
          visiter k;
          traiter reste (!cfc :: liste_CFC)
        end in
  traiter s1 []
```

## Solutions