**Université de Caen Normandie** **Année 2020 – 2021**
**UFR de Sciences**

**Autre Paradigme**
**Parties 2, 3 et 4 du DM**

**Nom et prénom : Nguyen Phuong Vy VU**
**L2 Informatique Groupe 2A**
**Nombre d'étudiant : 21911658**

## 2. Prétraitement : Suppression des tautologies

**Question 1.** Pourquoi peut-on supprimer d'une formule toutes les clauses qui sont des tautologies ?

On peut supprimer une formule tautologie $\top$ parce que la tautologie est l'élément neutre de conjonction, on a bien $\top \wedge A \equiv A \wedge \top = A$

**Question 2.** Donner une condition suffisante pour qu'une clause soit une tautologie. Cette condition est-elle nécessaire ?

La formule A est une tautologie si et seulement si $A \equiv T$. Cette condition est nécessaire, elle peut simplifier une formule.

**Question 3.** Définir la fonction `(negation lit)` qui à un littéral associe sa négation.

```
negation :: Litteral -> Litteral
negation (Pos a) = Neg a
negation (Neg a) = Pos a
```

**Question 4.** Définir la fonction `(estTauto ls)`.

```
estTauto :: Clause -> Bool
estTauto ls = and [or [valeur lit dist | lit <- ls ] | dist <-
   genereTable (rmdups (listeVarsClause ls))]
```

## 3. Existence d'une distribution : mise en œuvre de DPLL (v1)

### 3.1. Conditions d'arrêt

**Question 1.** Justifier la seconde condition d'arrêt.

On sait qu'une formule est évidemment contradictoire si et seulement si elle contient deux clauses unitaires de la forme `litt` et négation de `litt`.

Alors, on a toujours $a \wedge \neg a = \bot$ et $\bot$ est l'élément absorbant d'une conjonction, on possède $\bot \wedge A \equiv \bot$. On peut conclure si $f$ est évidemment contradictoire alors $f$ est insatisfaisable.

**Question 2.** Définir la fonction `(estUnitaire c)` qui détermine si une clause `c` est unitaire.

```
estUnitaire :: Clause -> Bool
estUnitaire c
    | length c == 1 = True
    | otherwise = False
```

**Question 3.** Compléter la définition de la fonction `(estEvidtContradic cs)`.

```
estEvidtContradic :: Formule -> Bool
estEvidtContradic [] = False
estEvidtContradic cs = chercher (clause cs)
    where chercher [] = False
          chercher (c:cs)
              | negation c `elem` cs = True
              | otherwise = chercher cs
          clause [] = []
          clause (c:cs)
              | estUnitaire c = c ++ clause cs
              | otherwise = clause cs
```

### 3.2. Simplification : règle du littéral seul (ou règle de la clause unitaire)

**Question 1.** Montrer que cette règle de simplification est correcte.

Pour montrer cette règle, on va trouver est-ce qu'une formule sera satisfiable (avoir au moins l'une de ses valeurs est *vraie*) après avoir appliqué cette règle. Soit la formule $f = (a \vee b) \wedge (\neg a) \wedge (a \vee \neg b \vee c)$ et on sait que $f$ est satisfiable.

Appliquer la règle de la clause unitaire, on a :

| | |
|---|---|
| Clause unitaire : a, donc $f = b \wedge (\neg b \vee c)$ | est satisfiable |
| Clause unitaire : b, donc $f = c$ | est satisfiable |
| Clause unitaire : c, donc $f = \emptyset$ | est satisfiable |

On obtient la formule satisfiable à la fin lorsque la formule de départ est satisfiable aussi. Donc, cette règle de simplification est correcte.

**Question 2.** Définir les fonctions `(existeSeul cs)` et `(trouveSeul cs)`.

```
existeSeul :: Formule -> Bool
existeSeul [] = False
existeSeul (c:cs)
    | estUnitaire c = True
    | otherwise = existeSeul cs

trouveSeul :: Formule -> Litteral
trouveSeul [] = error "Pas trouve"
trouveSeul (c:cs)
    | estUnitaire c = head c
    | otherwise = trouveSeul cs
```

**Question 3.** Définir la fonction `(elimineSeul cs lit)` qui applique cette règle de simplification.

```
elimineSeul [] _ = []
elimineSeul (c:cs) lit
    | lit `elem` c = elimineSeul cs lit
    | negation lit `elem` c = elimineLit c (negation lit) :
       elimineSeul cs lit
    | otherwise = c : elimineSeul cs lit
        where elimineLit [] _ = []
              elimineLit (c:cs) lit
                | c == lit = elimineLit cs lit
                | otherwise = c : elimineLit cs lit
```

### 3.3. Simplification : règle du littéral pur

**Question 1.** Montrer que cette règle de simplification est correcte.

On va chercher la satisfiabilité d'une formule qui contient une clause pure (avoir un littéral pur). Soit $f = (b \vee \neg a) \wedge (c \vee \neg b) \wedge (\neg a \vee b \vee \neg c) \wedge (b \vee c)$ dont $\neg a$ est un littéral pur et $f$ est satisfiable.

| | |
|---|---|
| Littéral pur : $\neg a$, donc $f = (c \vee \neg b) \wedge (b \vee c)$ | est satisfiable |
| Littéral pur : c, donc $f = \emptyset$ | est satisfiable |

On peut éliminer les clauses pures de $f$ sans changer sa satisfiabilité. Donc, cette règle de simplification est correcte.

**Question 2.** En vous inspirant du travail mené pour mettre en œuvre la règle du littéral seul (cf Section), définir les fonctions `(existePur cs)` et `(trouvePur cs)` ainsi que la fonction `(eliminePur cs lit)`.

```
existePur :: Formule -> Bool
existePur cs = existe (rmdups (concat cs))
    where existe [] = False
          existe (c:cs)
            | negation c `notElem` cs = True
            | otherwise = existe (filter (/= negation c) cs)

trouvePur :: Formule -> Litteral
trouvePur [] = error "Pas trouve"
trouvePur cs = head (supprimer (rmdups (concat cs)))
    where supprimer [] = []
          supprimer (c:cs)
            | negation c `notElem` cs = c : supprimer cs
            | otherwise = supprimer (filter (/= negation c) cs)

eliminePur :: Formule -> Litteral -> Formule
eliminePur [] _ = []
eliminePur (c:cs) lit
    | lit `elem` c = eliminePur cs lit
    | otherwise = c : eliminePur cs lit
```

### 3.4. Recherche exhaustive (splitting)

**Question 1.** Justifier l'architecture de la fonction `(estSatis f)`, i.e. en quoi applique-t-elle correctement `DPLL` ?

La fonction `estSatis f` applique `DPLL` en éliminant les clauses unitaires et les littérals purs avec les règles de simplification. S'il n'y les a pas, elle choisit une clause unitaire qui est le premier littéral ou la négation du premier littéral de la formule et répéter les étapes d'éliminer avec la règle de la clause unitaire jusqu'à la formule $f$ est vide donc elle est satisfiable. Mais si la formule à la fin est évidemment contradictoire, alors elle n'est pas satisfiable.

**Question 2.** Compléter la définition de la fonction `(estSatis f)`.

```
estSatis :: Formule -> Bool
estSatis [] = True
estSatis f
    | estEvidtContradic f = False
    | existeSeul f = estSatis (elimineSeul f (trouveSeul f))
    | existePur f = estSatis (eliminePur f (trouvePur f))
    | otherwise = estSatis (elimineSeul f (head (concat f))) ||
        estSatis (elimineSeul f (negation (head (concat f))))
```

**Question 3.** Pourquoi n'applique-t-on l'élimination des tautologies qu'une fois et une seule sous forme de prétraitement ?

Dans l'application de la règle de la clause unitaire, si la formule à la fin est tautologie, alors la formule de départ ne peut pas être une tautologie. Cela affecte le résultat donc on doit éliminer les tautologies avant de traiter une formule.