DPLL Algorithm for solving the CNF-SAT problem.

Manipulation de formules

Hauteur d'une formule

Soit ϕ une formule définit comme suivant:

- Les atomes
  - Toutologie T
  - Absurde $\bot$
  - variables propositionnelles: a, b, c, ...
- Les formules composites:
  - Conjonction ϕ Et ψ (ψ, ϕ étant deux formules)
  - Disjonction ϕ Ou ψ (ψ, ϕ étant deux formules)
  - Négation $\neg\phi$

La hauteur d'une formule ϕ est définit de façon récursive, à savoir:

- si ϕ est un atome(T ou $\bot$ ou variable), alors `hauteur(ϕ) = 0`
- si ϕ est une composition, alors:
  - hauteur(Non ϕ) = `hauteur(ϕ) + 1`
  - hauteur(ϕ op ψ) = `max(hauteur(ϕ), hauteur(ψ)) + 1` (avec op $\in \{\wedge, v\}$)

Les formules de type implication et équivalence ne sont pas prisent en charge dans notre implémentation(en suivant l'énnoncé du projet).

En se basant sur cette définition, notre implémentation reprend les mêmes étapes citées en dessus.

```
-- >>> hauteur ((Var "b" `Et` Var "c") `Et` Var "a")
-- 2
hauteur :: Formule -> Int
hauteur Vrai    = 0                                    -- cas d un atome: T
hauteur Faux    = 0                                    -- cas d un atome: ⊥
hauteur (Var v) = 0                                    -- cas d un atome: Variable pro
hauteur (Non f) = hauteur f + 1                        -- cas d une composition: negat
hauteur (Et f1 f2) = max (hauteur f1) (hauteur f2) + 1 -- cas d une composition: conjo
hauteur (Ou f1 f2) = max (hauteur f1) (hauteur f2) + 1 -- cas d une composition: disjc
```

Ainsi en prenant un exempe de formule: ((c ∧ b) ∧ a), la fonction calcule la hauteur comme suivant:

> Ce qui donne: hauteur(((c ∧ b) ∧ a)) = 2

Variables d'une formule

En partant sur la même logique, on peut calculer l'ensemble des variables d'une formule d'une façon inductive(récursive).

Cette fois-ci on doit construire un ensemble contenant toutes les variables d'une formule. Cette ensemble sera construit récursivement de la façon suivante:

- si ϕ est une tautologie ou absurde, alors `Var(ϕ) = {ensemble vide}`
- si ϕ est une variable props.(nommée x), alors `Var(ϕ) = {x}`
- si ϕ est sous forme d'une négation, alors `Var(Non ϕ) = Var(ϕ)` . càd on continue notre recherche sur la sous formule
- si ϕ est une composition(conjonction ou disjonction), alors `Var(ϕ op ψ) = Var(ϕ) Union Var(ψ) (avec op ∈ {∧, ∨})` . càd on continue notre recherche sur les deux sous formules et on réalise un union des deux sous-ensembles.

Et c'est exactement ce que fait notre implémentation.

```
-- >>> variables (((Var "c" `Et` Var "b") `Et` (Var "a")))
-- fromList ["a","b","c"]
variables :: Formule -> Set String
variables Faux = Set.empty                                  --  cas tautologie
variables Vrai = Set.empty                                  --  cas absurde
variables (Var v)    = Set.singleton v                      --  cas variable props.
variables (Non f)    = variables f                          --  cas d une formule sou
variables (Et f1 f2) = variables f1 `Set.union` variables f2 --  cas d une formule con
variables (Ou f1 f2) = variables f1 `Set.union` variables f2 --  cas d une formule dis
```

Ainsi, en reprenant le même exemple précédent: ((c ∧ b) ∧ a) La fonction contruit l'ensemble comme suivant:

> Ce qui donne: ({b} Union {c}) Union {a} = {b, c, a}

Évaluation d'une formule

Maintenant on veut évaluer une formule ϕ en se basant sur un environnement d'évaluation qu'on va nommé `sigma` . L'environnement `sigma` associe à chaque variable de la formule une valeur logique(Vrai ou Faux), ainsi l'évaluation de toute la formule se fera de façon inductive à savoir:

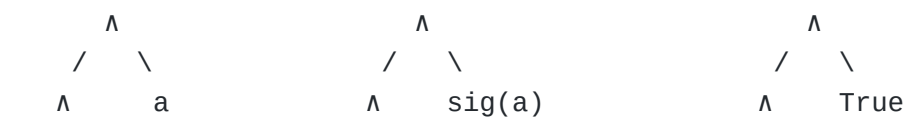
- si ϕ est (respectivement) un T ou ⊥, alors `Val(ϕ, sigma) = Vrai` ou `Val(ϕ, sigma) = Faux`
- si ϕ est une variable props.(nommée x), alors `Val(ϕ, sigma) = sigma(x)`
- si ϕ est sous forme d'une négation, alors `Val(Non ϕ, sigma) = Non Val(ϕ)` . Càd qu'on continue l'évaluation sur la sous formule, mais aussi on applique la négation sur son résultat.
- si ϕ est sous forme d'une conjonction, alors `Val(ϕ op ψ, sigma) = Val(ϕ) op Val(ψ)` (avec op ∈ {∧, v}). càd qu'on continue l'évaluation sur les 2 sous formules et on applique l'opération (Ou, Et) sur le résultat des 2 valuations.

En haskell, cela peut se traduire comme:

```
sigma :: Valuation
sigma = Map.fromList [("a", True), ("b", True), ("c", False)]

-- >>> evaluate sigma (((Var "c" `Et` Var "b") `Et` (Var "a")))
-- True
evaluate :: Valuation -> Formule -> Bool
evaluate _ Vrai = True
evaluate _ Faux = False
evaluate v (Var var)  = v ! var
evaluate v (Non f)    = not (evaluate v f)
evaluate v (Et f1 f2) = evaluate v f1 && evaluate v f2
evaluate v (Ou f1 f2) = evaluate v f1 || evaluate v f2
```

En prenant le même exemple précédent, la fonction évalue une formule comme suivant:



```
     /   \              /   \                  /   \
    c     b         sig(c) sig(b)           False  True
```

> Ce qui donne: Val(((c ∧ b) ∧ a)) = ((True ∧ False) ∧ True) = False

Mise sous forme normale conjonctive

Pousser les négations sur les variables

A cette étape, on doit pousser les négations sur les variables, càd on doit transformer la formule originale en une formule équivalente mais dont les négations ne portent que sur des variables. Pour faire cela on aura besoin d'utiliser quelques règles:

- i. Non Vrai = Faux
- ii. Non Faux = Vrai
- iii. double négation: Non (non a) = a
- iv. loi de Morgan:
  - Non(a Et b) = (Non a) Ou (Non b)
  - Non(a Ou b) = (Non a) Et (Non b)

Et comme vu précédemment, une formule est définit inductivement. Donc si l'on veut pousser les négations sur les variables il faut procéder selon la forme de la formule, ainsi dans chaque cas on applique une des règles citée en dessus.

Soit ϕ une formule, on peut distinguer plusieur formes:

- si ϕ est une négation de tautologie(`Non T`), alors `Push(ϕ) = ⊥` . Application de la règle 1
- si ϕ est une négation de tautologie(`Non ⊥`), alors `Push(ϕ) = T` . Application de la règle 2
- si ϕ est une double négation (`Non(Non ϕ)`), alors `Push(ϕ) = Push(ϕ)` . Dans ce cas là, on applique la règle 3 et on continu l'exploration de la sous formule ϕ
- si ϕ est sous forme `Non(ϕ op ψ)` , alors `Push(Non(ϕ op ψ)) = Push(Non ϕ) op Push(Non ψ)` (avec op ∈ {∧, v}). Dans ce cas, on applique les lois de Morgan comme indiqué ci-dessus, et on continue l'exploration des 2 sous formules(ϕ, ψ).
- Pour les autres formes que peut prendre ϕ, par exemple :
  - si ϕ est une conjonction(`ϕ Et ψ`), alors `Push(ϕ Et ψ) = Push(ϕ) Et Push(ψ)` . Tout simplement, on continue l'exploration des 2 sous formules (ϕ, ψ)
  - (même chose) si ϕ est une disjonction(`ϕ Ou ψ`), alors `Push(ϕ Ou ψ) = Push(ϕ) Ou Push(ψ)` .

Notre implémentation reprend le même enchainement décrit en haut.

```
-- >>> pushNegation (Non (Var "a" `Ou` (Non (Var "b") `Et` Var "c" )))
-- (¬a ∧ (b v ¬c))
pushNegation :: Formule -> Formule
pushNegation f =
    case f of
        -- Non tautologie
        Non Vrai       -> Faux
        -- Non absurde
        Non Faux       -> Vrai
        -- double negation
        Non (Non f)   -> pushNegation f
        -- lois de Morgan
        Non (Et f1 f2) -> pushNegation (Non f1) `Ou` pushNegation (Non f2)
        Non (Ou f1 f2) -> pushNegation (Non f1) `Et` pushNegation (Non f2)
        -- explorer les sous formules
        Et f1 f2       -> Et (pushNegation f1) (pushNegation f2)
        Ou f1 f2       -> Ou (pushNegation f1) (pushNegation f2)
        -- le reste
        f1             -> f1
```

Par exemple si l'on veut pousser les négations sur la formules: ¬(a v (b ∧ c)), on peut remarquer que la formule est de type Non(ϕ Ou ψ) avec ϕ = Var(a) et ψ = (b ∧ c), donc en appliquant les lois de Morgan ca se simplifie en ¬a ∧ (b v ¬c).

> Autre exemple:
>
> - ϕ = Non(Vrai) se simplifie simplment en Faux
> - ϕ = ¬(a ∧ (b v c)) se simplifie en ¬a v (b ∧ ¬c). (en appliquant la loi de morgan)
> - ϕ = ¬(¬ (a v b)) se simplifie en (a v b). (règle de double négation)

Calculer la forme normale conjonctive

La distributivité de 'Ou' par rapport à 'Et'

La règle de distributivité de "Ou" par rapport à "Et" permet une simplification de la formule, mais aussi elle aide à rendre la formule en forme normal conjonctive.

Dons le sujet, la ditributivité va s'appliquer entre deux sous formules supposées en forme normal conjonctives en appliquant le "Ou" entre les deux. Ainsi, on aura besoin à appliquer les règles communes de la distributivité qu'on va citer:

- i. (a ∧ b) v (c ∧ d) = (a v c) ∧ (a v d) ∧ (b v c) ∧ (b v d)
- ii. a v (b ∧ c) = (a v b) ∧ (a v c)
- iii. (a ∧ b) v c = (a v c) ∧ (b v c)

Il existe différent cas possible selon la forme de la formule, et en utilisant les règles en dessus on peut les déterminer:

- si ϕ est de type `(f1 Et f2) Ou (f3 Et f4)`, alors Dist(ϕ) = (f1 Ou f3) `Et` (f1 Ou f4) `Et` (f2 Ou f3) `Et` (f2 Ou f4). En appliquant la 1ère règle.
- si ϕ est de type `f1 Ou (f2 Et f3)`, alors Dist(ϕ) = (f1 Ou f2) `Et` (f1 Ou f3). En appliquant la règle (2) sur les 2 sous formules `f1` et `(f2 Et f3)`.
- si ϕ est de type `(f1 Et f2) Ou f3`, alors Dist(ϕ) = (f1 Ou f3) `Et` (f2 Ou f3). En appliquant la règle (3) sur les 2 sous formules `(f1 Et f2)` et `f3`.

On peut aussi prendre en compte les autre formes de formule normale conjonctive:

- si ϕ est de type `(f1 Ou f2) Ou (f3 Ou f4)`, alors Dist(ϕ) = ϕ. Puisque ϕ est constituée de deux sous formules en forme normale conjonctive. La première sous formule est `(f1 Ou f2)` et la seconde est `(f3 Ou f4)`. Donc, dans ce cas là la distibutivité ne s'applique pas.
- On peut avoir deux autres cas similaire, à savoir:
  - ϕ est de type `(f1 Ou f2) Ou (Var v)`, alors Dist(ϕ) = ϕ. Pour la même raison précédente. Les deux sous formules sont des formules normal conjonctive mais auxquel on peut pas appliquer la ditributivité.
  - ϕ est de type `(Var v) Ou (f1 Ou f2)`, Dist(ϕ) = ϕ. La même chose.

Dans notre implémentation, on a pu faire la même chose ce qu'on a expiqué en dessus.

```
-- >>> distribute ((Var "x" `Ou` Var "y") `Et` (Non (Var "z") `Ou` (Var "x"))) ((Var "
-- (((x v y) v t) ∧ (((x v y) v (s v ¬y)) ∧ (((¬z v x) v t) ∧ ((¬z v x) v (s v ¬y)))))
distribute :: Formule -> Formule -> Formule
distribute (Et f1 f2) (Et f3 f4)                                    = Et (Ou f1 f3) (
distribute f1 (Et f2 f3)                                            = (f1 `Ou` f2) `E
distribute (Et f1 f2) f3                                            = (f1 `Ou` f3) `E
distribute disj1@(Ou f1 f2) disj2@(Ou f3 f4)                        = disj1' `Ou` dis
                                                                      where disj1' =
                                                                            disj2' =
distribute (Var v1) (Var v2)                                        = Var v1 `Ou` Var
distribute (Var v)  disj@(Ou f1 f2)                                 = Var v `Ou` disj
distribute disj@(Ou f1 f2) (Var v)                                  = disj `Ou` Var v
distribute f neg@(Non (Var v2))                                     = f `Ou` neg
distribute neg@(Non (Var v1)) f                                     = neg `Ou` f
```

De cette façon, on établie une implémentation générale quelque soit la forme des 2 sous formules conjonctives.

Conversion en FNC

La conversion en forme normal conjonctive se fait comme suivant:

- si ϕ est une `variable "v"` , alors `FNC(ϕ) = Var(v)`
- si ϕ est une conjonction de deux formules `f1 Et f2` , alors `FNC(ϕ) = FNC(f1) Et FNC(f2)` . Dans ce cas, on doit continuer à explorer les deux sous formules (f1, f2) en faisant un Et logique sur le résulat de l'exploration
- si ϕ est une disjonction `f1 Ou f2` , alors `FNC(ϕ) = Dist(FNC(Push(f1)) FNC(Push(f2)))` . Dans ce cas, comme c'est une dijonction alors on est obligé de passer par pousser la négation et par la distribution afin de calculer la forme normale conjonctive associée à ce type de formule. Lorsqu'on pousse la négation sur les 2 sous formules(f1, f2), on doit re-explorer les 2 sous formules résultantes avant d'effectuer la distribution. Cela nous permettra de redéterminer la nouvelle forme des deux sous formules f1, f2, ainsi on peut effectuer la distribution sans problème.
- si ϕ est sous forme de négation `Non(ϕ)` , alors `FNC(ϕ) = Push(Non (FNC(ϕ)))` . Tout simplement, on doit pousser la négation que lorsqu'on a obtenu la forme normale conjonctive de ϕ.

En haskell, cela peut d'écrire comme:

```
fnc :: Formule -> Formule
fnc litteral@(Var v) = litteral
fnc (Et f1 f2)       = fnc f1 `Et` fnc f2
fnc (Ou f1 f2)       = distribute (fnc (pushNegation f1)) (fnc (pushNegation f2))
fnc (Non f)          = pushNegation (Non $ fnc f)
```

> Par exemple: (la forme normale conjonctive équivalente est en jaune)
>
> - -- >>> fnc ((A ∧ B) v (C ∧ D))
>   - `((A v C) ∧ ((A v D) ∧ ((B v C) ∧ (B v D))))`
> - -- >>> fnc (((A v B) v E) v (C v D))
>   - `(((A v B) v E) v (C v D))`
> - -- >>> fnc (((A ∧ X) v B) v (C v D))
>   - `(((A v B) v (C v D)) ∧ ((X v B) v (C v D)))`
> - -- >>> fnc ((A v X) ∧ (B ∧ (C v D)))
>   - `((A v X) ∧ (B ∧ (C v D)))`

Une représentation des problèmes SAT

La gestion des variables

Afin de compiler une formule de la logique propositionnelle en un problème SAT, on a besoin d'encoder les variables de la formule par des nombres entiers strictement positif. Pour faire cela, on

peut construire un dictionnaire dont les clés représentent des variables et les valeurs représentent les entiers associés à ces variables. Inversement, on peut décoder les variables en construisant un dictionnaire inverse dont les clés sont les entiers et les valeurs sont des variables associes à ces entiers.

En haskell, on peut le faire simplement avec la notion de liste en compréhension. Cette liste contiendra des couples qui représentent (clé, valeur). Ainsi, en parcourant l'ensemble des variables d'une formule on peut construire cette liste en attribuant à chaque variable parcouru un entier k (avec k > 0 et k ∈ [1, nombreVariables]). On peut faire la même chose avec le dictionnaire "decode".

Par conséquent, l'implémentation suivante reprend ce qu'on a expliqué:

```
-- >>> varsToInt (Set.fromList ["a", "b"])
-- (fromList [("a",1),("b",2)],fromList [(1,"a"),(2,"b")])
varsToInt :: Set String -> (Map String Int, Map Int String)
varsToInt s = (encode, decode)
              where
                encode = Map.fromList[(Set.toList s !! (i-1), i) | i <- [1 .. size s]]
                decode = Map.fromList[(i, Set.toList s !! (i-1)) | i <- [1 .. size s]]
```

```
-- >>> varRepresentation (((Var "a" `Et` Var "b") `Ou` Faux) `Et` (Non (Var "c") `Et`
-- (fromList [("a",1),("b",2),("c",3)],fromList [(1,"a"),(2,"b"),(3,"c")])
varRepresentation :: Formule -> (Map String Int, Map Int String)
varRepresentation f = varsToInt (variables f)
```

La compilation vers SAT d'une formule sous forme normale conjonctive

- Dans cette partie on représente les variables par des entiers strictement posistifs: par exemple `K` représente la variable `Xk`
- le littéral ¬Xk est représenté par l'entier -K

Ainsi, On représente un problème SAT par un ensemble d'ensemble d'entier `sat` = {C1, ..., Cn} .

- Chaque avec Ci est un ensemble d'entiers représentant une clause. Par exemple l'ensemble {1, -2, 3} représente la clause (a v b v c)
- L'ensemble `sat` représente la conjonction de ces clauses

Le but étant donné une formule on doit construire l'ensemble `sat` correspondant. Pour faire cela, on doit d'abord convertir notre formule en FNC, ensuite on distingue les cas suivants:

- Cas de base:
  - si la formule `F` est qu'une variable `X`, dans ce cas on construit un ensemble qui contient une clause unitaire. càd `SAT(F) = {{1}}` en supposant l'association `("X", 1)`.
  - si la formule est une negation d'une variable `(Non Var a)`, dans ce cas on construit un ensemble qui contient l'opposé de cette variables. càd `SAT(F) = {{-1}}`.
- Cas génaral:
  - si la formule `F` est une conjonction `(f1 Et f2)`, on continue l'exploration sur les 2 sous formules et on réalise l'union des 2 clauses. Autrement, `SAT(F) = SAT(f1) U SAT(f2)`.
  - si la formule `F` est une disjonction `(f1 Ou f2)`, on continue l'exploration des deux sous formules et on essaye de rassembler le résultat dans une seule clause(car se sont des variables séparées par des `Ou`). Autrement, `SAT(F) = {[SAT(f1) ++ SAT(f2)]}`.

Et c'est ce qu'il fait le code suivant:

```
fncToSAT :: Map String Int -> Formule -> SAT
fncToSAT m f =  case f of
                    Var v  -> Set.fromList [Set.fromList[m ! v]]                 -- Ca
                    Non (Var v) ->  Set.fromList [Set.fromList[-(m ! v)]]        -- Ca
                    Et x y -> fncToSAT m x `union` fncToSAT m y                  -- Ca
                    Ou x y -> Set.fromList [Set.fromList (convOu x ++ convOu y)] -- Ca
                where
                    convOu (Ou x y) = convOu x ++ convOu y
                    convOu (Var v)  = [m ! v]
                    convOu (Non (Var v)) = [-(m ! v)]
```

Et la fonction principal `formuleToSat` permet d'abord de traduire en FNC la formule passé en argument et convertit la formule en un problème SAT en traduisant les variables en nombre entier.

```
formuleToSAT :: Formule -> SAT
formuleToSAT f = fncToSAT (fst(varRepresentation f)) (fnc f)
```

> Voici des exemples d'execution:
>
> - i. si f = a, alors `formuleToSAT(f) = {{1}}`
> - ii. Si f = ((a v b) ∧ c), alors `formuleToSAT(f) = {{1, 2}, {3}}`
> - iii. Si f = (¬a ∧ (b v ¬c)), alors `formuleToSAT(f) = {{-3, 2}, {-1}}`
> - iv. si f = ((a v b) ∧ (¬c v a)), alors `formuleToSAT(f) = {{-3, 1}, {1, 2}}`
>
> Les mêmes exemples en haskell:

```
-- Exemple 1.
>>> f = (Var "a")
>>> formuleToSAT f
```

```
    fromList [fromList [1]]

    -- Exemple 2.
    >>> f = (Var "a" `Ou` Var "b") `Et` Var "c"
    >>> formuleToSAT f
    fromList [fromList [1,2],fromList [3]]

    -- Exemple 3.
    >>> f = (Non (Var "a") `Et` (Var "b" `Ou` Non (Var "c")))
    >>> formuleToSAT f
    fromList [fromList [-3,2],fromList [-1]]

    -- Exemple 4.
    >>> f = ((Var "a" `Ou` Var "b") `Et` (Non (Var "c") `Ou` (Var "a")))
    >>> formuleToSAT f
    fromList [fromList [-3,1],fromList [1,2]]
```

À propos de la FNC

Un cas où une formule dont la FNC possède une taille exponentielle : F1 = (X1 ∧ Y1) v (X2, Y2) v ... v (Xn ∧ Yn)

-> la FNC correspondante est de taille 2**n est de la forme: (X1 v ... v X(n-1) v Xn) ∧ (X1 v ... v Xn-1 v Yn) ∧ ... ∧ (Y1 v ... v Y(n-1) v Yn)

Pour éviter les transformations exponentielles, il est possible d'appliquer des transformations en introduisant des variables supplémentaires. Par conséquent, ce type de transformation ne crée plus des formules logiquement équivalentes, mais des transformations qui préservent la satisfiabilité de la formule originale.

La formule F1 en haut peut être récrite en introduisant les variables Z1, ..., Zn

-> (Z1 v ... v Zn) ∧ (-Z1 v X1) ∧ (-Z1 v Y1) ∧ ... ∧ (-Zn v Xn) ∧ (-Z v Yn)

Dans cet exemple, la variable Zi représente la vérité de la i-ème conjonction de la formule originale. Si Zi est vraie, alors Xi et Yi doivent être vraies aussi. La première clause de la transformation impose qu'au moins un des Zi soit vrai pour que la formule soit satisfaite, donc qu'au moins une des clauses de la formule originale soit vraie.

Une telle transformations permettent d'obtenir une formule en FNC dont la taille est linéaire par rapport à la taille de la formule originale.

Implémenter DPLL sur les problèmes SAT

L'opération de simplification

A ce stade, on doit pouvoir éliminer toutes les clauses triviales(qui contienent à la foir un nombre K et son opposé -K) du problème SAT. Pour faire cela, on parcourt les clauses de notre problème SAT et on filtre les clauses triviales des clauses qui le sont pas. En haskell, la fonction `filter` permet de filtrer un ensemble SAT en lui donnant comme paramètre le filtre `les clauses qui ne sont pas triviales`

```
simplification :: SAT -> SAT
simplification  = Set.filter (not . isTrivial)
```

Le prédicat isTrivial permet de vérifier pour chaque nombre de la clause s'il y en a son opposé, si oui renvoie vrai(la clause est triviale), sinon faux. Ainsi, il est determiné comme:

```
-- >>> isTrivial (Set.fromList[1, 3])
-- False
-- >>> isTrivial (Set.fromList[1, 2, 3, -2])
-- True
isTrivial :: Clause -> Bool
isTrivial c = any (\x -> (-x) `member` c) c
```

-> Note: Si un problème SAT ne contient pas de clause trivial alors dans ce cas on renvoie le même ensemble.

> Exemples de simplification:
>
> - si f = ((a v ¬a) ∧ b), alors simplification(f) = {{2}} (avec l'encodage [("a", 1), ("b", 2)]. On a éliminé la clause (a v -a) car elle est triviale, donc reste la clause {b}
>
> Voici le même exemple en haskell:

```
>>> f = ((Non (Var "a") `Ou` Var "a") `Et` Var "b")
>>> sat = formuleToSAT f
>>> simplification sat
fromList [fromList [2]]
```

Propagation de valeur

La propagation d'une valeur `K` permet d'éliminer du problème SAT les clauses contenant cette valeur, et simplifie les clauses contenant l'opposé `-K` en le supprimant de cette clause.

Pour implémenter cela, l'idée est de filtrer les clauses qcontenant le nombre `K` des clauses qui le possède pas, ensuite on procède sur chacunes des clauses qui restent en éléminant que l'opposé `-K`.

En haskell, on peut faire ça en utilisant la fonction map sur l'ensemble SAT après l'élimination des clauses contenant `K`, et on applique la fonction(1èr argument de map) qui élimine les `-K` des clauses.

```
propagate :: Int -> SAT -> SAT
propagate k s = Set.map ( Set.filter (/= -k)) (Set.filter(not . member k) s )
```

> Par exemple: si f = ((a v ¬b) ∧ b), alors `propagate(2, f) = {{1}}` (avec le même encodage précédent a->1, b->2). On a éliminer la clause contenant b ensuite on a éliminer (-b) de la clause (a v -b), donc il reste que {{a}} <=> {{1}}
>
> Même exemple en haskell:
>
> ```
> >>> f = (Var "a" `Ou` (Non(Var "b"))) `Et` ((Var "b"))
> >>> sat = formuleToSAT f
> >>> propagate 2 sat
> fromList [fromList [1]]
> ```

Clauses unitaires et litéraux purs

1. Clauses unitaires

Cette partie est facile, on doit reprérer les clauses qui contiennent qu'un seul littéral, ensuite on doit pouvoir stocker ces littéreaux dans un ensemble d'entiers.

En haskell, cela peut se traduire tout simplement comme:

- la fonction suivante permet de conserver que les clauses unitaires contenus dans le problème SAT, ensuite elle essaye de regrouper tous les littéreaux contenus dans ces cluases dans un ensemble d'entiers.

```
literalsInUnitaryClauses :: SAT -> Set Int
literalsInUnitaryClauses s = literals (Set.filter isUnitary s)
```

- Le prédicat `isUnitary(clause)` permet de determiner si une clause est unitaire(càd si elle est de taille 1).

```
isUnitary :: Clause -> Bool
isUnitary c = Set.size c == 1
```

> **Exemple**: F = ((a v ¬b) ∧ b) ∧ c, en prenant l'encodage suivant: a->1, b->2, c->3
>
> ```
> >>> F = ((Var "a" `Ou` (Non(Var "b"))) `Et` (Var "b")) `Et` (Var "c")
> >>> sat = formuleToSAT F
> >>> literalsInUnitaryClauses sat
> fromList [2,3]
> ```
>
> **Explication du résultat**: sur la formule `F` on remarque que b et c sont 2 clauses unitaires, donc le résultat est l'ensemble {b, c} -> {2, 3}

2. Problème SAT contradictoire

Dire un problème SAT est contradictoire c'est aussi dire que 2 clauses unitaires contienent des littéraux opposés. Donc on peut simplement extraire l'ensemble des littéraux des clauses unitaires et vérifier si pour chaque nombre de cet ensemble il existe au moins son opposé. Si c'est le cas, alors le problème est contradictoire, sinon il l'est pas.

En haskell on peut utiliser la fonction `any` qui va permettre de vérifier s'il existe au moins un littéral opposé dans l'ensemble des littéraux des clauses untaires. l'ensemble des littéraux des clauses untaires est renvoyé par la fonction `literalsInUnitaryClauses sat` vu précédement.

```
contradiction :: SAT -> Bool
contradiction s = any (\x -> (-x) `member` literalsInUnitaryClauses s) (literalsInUnit
```

> Exemple: avec le même encodage de a et b
>
> - i. F = ((a v ¬b) ∧ b) ∧ ¬b -> True (car le problème SAT contient 2 clauses unitaires opposées (b ∧ ¬b), ce qui est contradictoire)
> - ii. F = (a v ¬b) ∧ b) -> False (ce cas n'apparaît pas de clauses contradictoire)
>
> ```
> >-- Exemple 1.
> >>> F = ((Var "a" `Ou` (Non(Var "b"))) `Et` (Var "b")) `Et` (Var "b")
> >>> sat = formuleToSAT f
> >>> contradiction sat
> True
> -- Exemple 2.
> >>> F = ((Var "a" `Ou` (Non(Var "b"))) `Et` (Var "b"))
> >>> sat = formuleToSAT f
> ```

```
>>> contradiction sat
False
```

3. Litéraux purs

Les litéraux purs se sont des litéraux qui apparaissent toujours dans les clauses en forme positive ou en négative. Par conséquent, les clauses contenant ces litéraux sont éliminés du problème SAT.

Pour déterminer les litéraux purs, on peut extraire l'ensemble des litéraux du problème SAT et vérifier pour chaque nombre de cet ensemble s'il en existe son opposé. Si son opposé existe alors on écarte ce litéral, sinon on stocke ce litéral dans un ensemble de litéraux purs.

En haskell on peut facilement utiliser la fonction `filter` à laquelle on passe le filtre "si l'opposé d'un litéral n'apparaît pas dans l'ensemble de litéraux" et qui applique ce filtre sur l'ensemble des litéraux d'un problème SAT. C'est ce que montre le code suivant,

```
pureLiterals :: SAT -> Set Int
pureLiterals s =  Set.filter (\x -> notMember (-x) purs) purs
                  where
                     purs = literals s
```

> **Exemple:** toujours avec le même encodage pour a, b et c avec f = (((a v ¬b) ∧ ¬b) ∧ ((¬b v c) v ¬a)) -> purLiterals(f) = {-2, 3}, car on remarque dans f que seuls les litéraux b et c qui apparaissent respectivement en négatif(pour b) et en positif(pour c), donc l'ensemble des litéraux purs est {-b, c} -> {-2, 3}
>
> Même exemple en Haskell
>
> ```
> >>> f = ((Var "a" `Ou` (Non(Var "b"))) `Et` (Non(Var "b"))) `Et` ((Non(Var "b")) `
> >>> sat = formuleToSAT f
> >>> pureLiterals sat
> fromList [-2,3]
> ```

Implémenter DPLL

-> Explication de l'algorithme élaboré:

- Entrée: un problème `sat`
- Output: vrai si `sat` est satisfiable, ou faux s'il n'est pas satisfiable

```
DPLL(sat):
 1. if ø ∈ sat then return false;
 2. if sat.isempty() then return true;
 3. if ∃ clause unitaire {l} ∈ sat then
 4.   C1 <- élimine les clauses in sat contenant le literal l;
 5.   C2 <- élimine tous les literaux (-l) in C1 ;
 6.   return DPLL(C2);
 7. if ∃ literal pur (l) in sat then
 8.   C1 <- élimine les clauses in sat contenant le literal l;
 9.   return DPLL(C1);
10. pivot <- Choisir une variable in sat;
11. return DPLL(sat ∪ {{pivot}}) Et DPLL(sat ∪ {{-pivot}});
```

Ce pseudo-code résume l'idée de notre algoithme DPLL

En raison de sa récursivité arborescente (la distinction des cas à la ligne 11), l'algorithme DPLL est évidemment parallélisable en évaluant les deux appels récursifs en parallèle. On peut le voir si nous regardons le graphe d'exécution sur l'exemple:

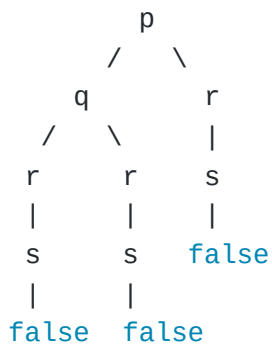
F = [(p v r), (p v s), (p v -r v -s), (-p v q r), (¬p, q , s), (¬p, q, ¬r, ¬s), (¬q, r), (¬q , s), (¬q, ¬r, ¬s)]

C'est un exemple compliqué, mais ce qui est interrésant c'est plûtot l'analyse des étapes réalisée par l'algo.

Voici l'arbre correspondant:

Un noeud affiche le littéral sélectionné à chaque étape (où le noeud racine est la première étape).

- Si un noeud n'a qu'un seul noeud enfant, il s'agit d'une exécution de la propagation unitaire.
- S'il a 2 noeuds enfants, alors c'est une exécution de la dernière règle(ligne 11). Dans ce cas, le bras gauche représente le chemin positif (c'est-à-dire que le littéral a été assigné vrai), et le bras droit représente le chemin négatif (c'est-à-dire que le littéral a été assigné faux).
- Les noeuds feuilles ont l'une des valeurs `true` ou `false` . Pour l'exemple, toutes les feuilles sont vraies, et donc la formule est insatisfaisante.
- Si l'arbre contient un chemin se terminant par une feuille marquée avec vrai, alors un modèle (partiel) de la formule (c'est-à-dire une affectation de vérité) peut être lu sur le chemin.



Vous allez remarquer que l'algo renvoie un booléan (Vrai si sat est satisfiable, sinon faux), mais il est tout à fait facile de le modifier càd de construire un ensemble d'entiers qui considérés comme vrais(satisfont le problème).

C'est ce que fait le code Haskell suivant, qui se base sur l'algo en haut:

```
dpllAux :: Set Int -> SAT -> Maybe(Set Int)
dpllAux sol sat
    | Set.null sat     = Just sol
    | any Set.null sat = Nothing
    | otherwise =
        case Set.toList(literalsInUnitaryClauses sat)  of
          (x:xs) -> dpllAux (Set.insert x sol) (propagate x sat)
          []     ->
            let pivot        = Set.findMin (literals sat)
                positiveCase = dpllAux (Set.insert pivot sol) (propagate pivot sat)
                negativeCase = dpllAux (Set.insert (-pivot) sol) (propagate (-pivot) s
            in case positiveCase of
              Nothing -> negativeCase
              Just xs -> Just xs
```

La fonction `dpll` confie tout le travaille à la fonction `dpllAux` qui implémente l'algo, mais avant tout elle permet de simplifier(élimination de clauses triviales) le problème SAT avant de commencer l'algo. De cette façon on s'assure qu'on utilise l'opération de simplification qu'une seule fois.

Ainsi, la fonction `dpll` permet de passer un ensemble d'entiers vide à la fonction `dpllAux` , cette dernière doit pouvoir le remplir et le renvoyer à la fin de l'algo (dans le cas oû le problème est satisfiable, sinon il va renvoyer un Nothing).

```
dpll :: SAT -> Maybe (Set Int)
dpll s = dpllAux Set.empty (simplification s)
```

> Exemple: f = (¬p v t) ∧ (p v ¬r) ∧ (q v r) ∧ (¬q v s) (avec p->1, q->2, s->3, r->4, t->5)
>
> Voici ce que l'execution affiche:

```
>>> f = ((Non(Var "p") `Ou` Var "t") `Et` (Var "p" `Ou` Non(Var "r")) `Et` (Var "c
>>> sat = formuleToSAT f
>>> dpll sat
Just (fromList [-3,-1,2,4])
```

Solve SAT-Problem

Après avoir parcouru tous les étapes de l'algorithme DPLL, on doit pouvoir résoudre notre problème en donnant une valuation possible(si c'est possible).

Comme on a déjà réalisé l'algo de DPLL, on pourrait simplement utiliser son résultat(un ensemble d'entiers(litéraux) qui satisfont le problème) puis de le parcourir pour déterminer une valuation pour chaque litéral de cet ensemble. Ainsi, on distingue 2 cas:

- si un litéral apparaît en négatif (-k) alors on va l'évaluer à False
- si un litéral apparaît en positif (k) alors on va l'évaluer à True
- si l'algo dpll a renvoyé un Nothing(formule contradictoire), alors automatiquement le Solver renverra un Nothing
- si l'algo dpll a renvoyé un ensemble non vide, alors le résultat de `solve` est un dictionnaire qui associe à chaque litéral (de l'ensemble des litéraux renvoyé par dpll) une valeur booléane(True ou False selon les cas cités en dessus)

Voici le code en Haskell qui reprend ce qu'on a expliqué:

```
solve :: Formule -> Maybe Valuation
solve f = let res = dpll (formuleToSAT f)
              rep = snd(varRepresentation f)
          in case res of
            Nothing -> Nothing
            Just s  -> Just (Map.fromList[Data.Bifunctor.first (rep !) (index (i - 1)
              where
                index :: Int -> [Int] -> (Int, Bool)
                index i s = if s !! i < 0
                              then (-(s !! i), False)
                            else
                              (s !! i, True)
```

> Exemple: si f = p ∧ (q v s) ∧ (¬q v ¬p) ∧ r (avec le même encodage), voici le résultat de l'execution
>
> ```
> >>> f = (Var "p" `Et` ((Var "q" `Ou` Var "s") `Et` (Non(Var "q") `Ou` Non(Var "p")
> >>> solve f
> Just (fromList [("p",True),("q",False),("r",True),("s",True)])
> ```

C'est à dire que les litéraux p, q, r et s satisfont le problème `f`, ainsi:

- `p s'évalue à True` car c'est un litéral positif dans l'ensemble renvoyé par dpll(même chose pour r et s)
- `q s'évalue à False` car c'est un litéral négatif dans l'ensemble renvoyé par dpll.