Quine

Dans ce sujet, on s'intéresse au problème SAT pour une formule quelconque, construite à partir de constantes, de variables et des connecteurs $\wedge, \vee$ et $\rightarrow$.

```
type formule =
  | C of bool
  | V of int (* entier positif ou nul *)
  | Et of formule * formule
  | Ou of formule * formule
  | Imp of formule * formule
  | Non of formule
```

On définit aussi le type suivant pour représenter des valuations :

```
type valuation = bool array
```

1 Algorithme en force brute pour SAT

1. On définit la taille $|f|$ d'une formule f comme le nombre total de nœuds (internes ou non) qu'elle contient. Écrire la fonction `taille`.

   ```
   taille : formule -> int
   ```

2. Écrire une fonction `var_max` telle que l'appel `var_max f` renvoie le plus grand entier i tel que la formule f contienne un nœud `V i`. On renverra -1 si f ne contient aucune variable.

   ```
   var_max : formule -> int
   ```

3. Écrire une fonction `evalue` qui prend en entrée une formule f et une valuation v et renvoyant $eval_v(f)$. On pourra supposer sans le vérifier que le tableau fourni est de longueur au moins `var_max f + 1`.

   ```
   evalue : formule -> valuation -> bool
   ```

4. À une valuation v de longueur n, on peut associer un entier $x = \sum_{i=0}^{n} v_i 2^i$ (en interprétant `true` comme 1 et `false` comme 0). Écrire une fonction `incremente_valuation` qui prend en entrée une valuation correspondant à un entier x et la modifie pour qu'elle corresponde après l'appel à l'entier $x + 1$. Si $x = 2^n - 1$, cette fonction lèvera l'exception `Derniere`, et le contenu du tableau après l'appel n'aura alors pas d'intérêt.

   ```
   exception Derniere
   incremente_valuation : valuation -> unit
   ```

5. Écrire une fonction `satisfiable_brute` qui détermine si une formule est satisfiable, en essayant toutes les valuations possibles jusqu'à les épuiser ou en trouver une convenable.

   ```
   satisfiable_brute : formule -> bool
   ```

6. Déterminer la complexité dans le pire cas de `satisfiable_brute`, on fonction de la taille $|f|$ de la formule f et de son nombre n de variables. On supposera que les variables de f sont numérotées consécutivement à partir de zéro.

2 Algorithme de Quine

1. Écrire une fonction `elimine_constantes` qui prend en entrée une formule f et renvoie une formule f' telle que $f' \equiv f$ et :

— soit f' est réduite à une constante ;
— soit f' ne contient aucune constante.
Dans la suite, on notera $s(f)$ la formule obtenue en appelant `elimine_constantes` sur une formule f.

```
elimine_constantes : formule -> formule
```

2. Écrire une fonction `substitue` telle que l'appel `substitue f i g` (où f et g sont des formules et i un entier) renvoie la formule $f[g/x_i]$.

```
substitue : formule -> int -> formule -> formule
```

3. On considère $f = \big(x_0 \to (x_1 \wedge (\neg x_0 \vee x_2))\big) \wedge \neg(x_0 \wedge x_2)$. Calculer $s(f[\top/x_0])$ et $s(f[\bot/x_0])$.

L'*algorithme de Quine* consiste, à partir d'une formule f, à calculer un *arbre de décision binaire* de la manière suivante :
— on commence par calculer $g = s(f)$;
— si g est une constante, l'arbre est réduit à une feuille, étiquetée par `true` ou `false` suivant la valeur de la constante ;
— sinon, on choisit une variable x apparaissant dans g et l'on calcule les arbres $a_\bot$ associé à $g[\bot/x]$ et $a_\top$ associé à $g[\top/x]$. L'arbre associé à f est alors $(i, a_\bot, a_\top)$.
L'arbre obtenu dépend du choix de la variable x à chaque étape.

4. Démontrer que cet algorithme termine.
5. À quelle condition sur l'arbre obtenu la formule de départ est-elle satisfiable ? une tautologie ?

On définit le type suivant pour les arbres de décision :

```
type decision =
  | Feuille of bool
  | Noeud of int * decision * decision
```

6. Écrire une fonction `construire_arbre` qui prend en entrée une formule f et renvoie un arbre de décision associé à f. On choisira à chaque étape la variable d'indice minimal apparaissant dans la formule.

```
construire_arbre : formule -> decision
```

7. Écrire une fonction `satisfiable_via_arbre` qui prend en entrée une formule et renvoie un booléen indiquant si elle est satisfiable, en construisant un arbre de décision.

```
satisfiable_via_arbre : formule -> bool
```

3 Un exemple d'application : le coloriage de graphes

On considère des graphes non orientés donnés sous la forme d'un tableau de listes d'adjacence :

```
type graphe = int list array
```

1. Pour un graphe G à n sommets et un entier k fixé, définir une formule $Col(G, k)$ qui soit satisfiable si et seulement si le graphe G est k-coloriable. On pourra utiliser des variables $(x_{i,c})_{0 \leqslant i < n, 0 \leqslant c < k}$ exprimant le fait que le sommet i est colorié avec la couleur c. Plusieurs solutions sont bien sûr possibles.
2. Écrire une fonction `encode` qui prend en entrée un graphe G et un entier k et renvoie la formule $Col(G, k)$.

```
encode : graphe -> int -> formule
```

3. Écrire une fonction `est_k_coloriable` tel que l'appel `est_k_coloriable g k` renvoie `true` si le graphe G est k-coloriable, `false` sinon. Cette fonction aura pour effet secondaire d'afficher le nombre de variables et la taille de la formule propositionnelle obtenue.

```
est_k_coloriable : graphe -> int -> bool
```

4. Écrire une fonction `chromatique` qui prend en entrée un graphe G et renvoie son nombre chromatique $\chi(G)$ (le plus petit entier k tel que G soit k-coloriable). On essaiera de limiter au maximum le nombre d'appels à `est_k_coloriable`.

Le format DIMACS est un format standard de description de graphes sous forme d'un fichier texte. Les fichiers fournis sont très simples :

— toute ligne commençant par un caractère `c` est un commentaire et doit être ignorée (ces lignes peuvent apparaître n'importe où dans le fichier) ;
— il y a une unique ligne commençant par `p edge` et contenant ensuite deux entiers (séparés par une espace) ; ces deux entiers indiquent respectivement le nombre de sommets et le nombre d'arêtes du graphe ;
— les autres lignes sont de la forme `e i j` (le caractère `e` suivi de deux entiers), et indiquent la présence d'une arête reliant le sommet i et le sommet j. Les lignes de cette forme apparaissent toutes après la ligne `p edge ...`.

Dans certains fichiers, une arête $\{i, j\}$ apparaît deux fois (une ligne `e i j` et une `e j i`), dans d'autres une seule fois. On fera en sorte de gérer correctement les deux cas.

5. Écrire une fonction `lire_dimacs` qui prend en entrée un nom de fichier au format DIMACS et renvoie le graphe correspondant.

```
lire_dimacs : string -> graphe
```

6. Calculer le nombre chromatique du graphe décrit par le fichier `myciel3.col`.

4 Version plus efficace

1. Écrire une fonction `simplifie` telle que l'appel `simplifie i b f` renvoie la formule $s(f[X/x_i])$ (où X vaut $\bot$ ou $\top$ suivant la valeur de `b`), par un calcul aussi efficace que possible. En particulier, on ne calculera pas $f[X/x_i]$ intégralement si on peut l'éviter.

```
simplfie : int -> bool -> formule -> formule
```

2. Écrire une fonction `satisfiable` qui détermine si une fonction est satisfiable, par le même principe que `satisfiable_via_arbre` mais sans construire explicitement l'arbre de décision.

```
satisfiable : formule -> bool
```

3. Déterminer le nombre chromatique du graphe décrit par `myciel4.col`, et si possible celui du graphe défini par `queen5_5.col`.