TP 32

Algorithme de 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.

II Algorithme de Quine

7. 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
```

8. 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
```

9. On considère $f = (x_0 \to (x_1 \wedge (\neg x_0 \vee x_2))) \wedge \neg(x_0 \wedge x_2)$. 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 $(i, a_\bot, a_\top)$.

Remarque. L'arbre obtenu dépend du choix de la variable x à chaque étape.

10. Démontrer que cet algorithme termine.
11. 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
```

12. 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
```

13. 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
```

III 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
```

14. 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 $(x_{i,c})_{0\leq i<n, 0\leq c<k}$ exprimant le fait que le sommet i est colorié avec la couleur c.

    Remarque. Plusieurs solutions sont bien sûr possibles.

15. 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
```

16. 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
```

17. 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`.

```
chromatique : graphe -> int
```

On retrouve le format DIMACS pour décrire des graphes sous forme d'un fichier texte (*cf.* TP 29). On redonne ci-dessous la fonction de lecture d'un tel fichier (pour décrire uniquement des graphes non orientés ici).

18. 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
```

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

```
let lire_dimacs fichier =
  let f_in = open_in fichier in
  let rec ligne_suivante () =
    let s = input_line f_in in
    if s.[0] = 'c' then ligne_suivante () else s
  in
  let n, _ =
    Scanf.sscanf (ligne_suivante ()) "p␣edge␣%d␣%d" (fun n p -> n, p)
  in
  let g = Array.make n [] in
  try
    while true do
      let i, j =
        Scanf.sscanf (ligne_suivante ()) "e␣%d␣%d" (fun i j ->
          i - 1, j - 1)
      in
      if not (List.mem j g.(i))
      then (
        g.(i) <- j :: g.(i);
        g.(j) <- i :: g.(j))
    done;
    assert false
  with
  | End_of_file -> g
```