TD XLIII

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

▶ **Question 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
```

▶ **Question 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
```

▶ **Question 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
```

▶ **Question 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
```

▸ **Question 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
```

▸ **Question 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

▸ **Question 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
```

▸ **Question 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
```

▸ **Question 9** 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 $(i, a_\bot, a_\top)$.

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

▸ **Question 10** Démontrer que cet algorithme termine.

▸ **Question 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
```

▸ **Question 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
```

▸ **Question 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
```

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
```

▸ **Question 14** Pour un graphe G à n sommets et un entier k fixé, définir une formule $\mathrm{Col}(G,k)$ 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.

Remarque

Plusieurs solutions sont bien sûr possibles.

▸ **Question 15** Écrire une fonction `encode` qui prend en entrée un graphe G et un entier k et renvoie la formule $\mathrm{Col}(G,k)$.

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

▸ **Question 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
```

▸ **Question 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`.

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 ...`.

Remarque

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.

▶ **Question 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
```

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

4 Version plus efficace

▶ **Question 20** 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
```

▶ **Question 21** 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
```

▶ **Question 22** Déterminer le nombre chromatique du graphe décrit par `myciel4.col`, et si possible celui du graphe défini par `queen5_5.col`.

Solutions

▶ Question 1

```
let rec taille f =
  match f with
  | C _ | V _ -> 1
  | Non f' -> 1 + taille f'
  | Et (f1, f2) | Ou (f1, f2) | Imp (f1, f2) -> 1 + taille f1 + taille f2
```

▶ Question 2

```
let rec var_max f =
  match f with
  | C _ -> -1
  | V i -> i
  | Et (f1, f2) | Ou (f1, f2) | Imp (f1, f2) -> max (var_max f1) (var_max f2)
  | Non f' -> var_max f'
```

▶ Question 3

```
let rec evalue formule valuation =
  match formule with
  | C b -> b
  | V i -> valuation.(i)
  | Et (f, g) -> (evalue f valuation) && (evalue g valuation )
  | Ou (f, g) -> (evalue f valuation) || (evalue g valuation)
  | Imp (f, g) -> (not (evalue f valuation)) || (evalue g valuation)
  | Non f -> not (evalue f valuation)
```

▶ Question 4

```
exception Derniere

let incremente_valuation valuation =
  let n = Array.length valuation in
  let i = ref (n - 1) in
  while !i >= 0 && valuation.(!i) do
    valuation.(!i) <- false;
    decr i
  done;
  if !i >= 0 then valuation.(!i) <- true
  else raise Derniere
```

▶ Question 5

```
let satisfiable_brute formule =
  let n = var_max formule in
  let valuation = Array.make (n + 1) false in
  try
    while not (evalue formule valuation) do
      incremente_valuation valuation
    done;
    true
  with
  | Derniere -> false
```

▶ **Question 6** Chaque appel à `evalue` se fait en temps proportionnel à $|f|$, chaque appel à `incremente` en temps $O(n)$ et donc $O(|f|)$. Au pire (si la formule n'est pas satisfiable, par exemple), il faut essayer les 2^n valuations possibles : on a donc une complexité en $\Theta(|f| \cdot 2^n)$ dans le pire cas.

▶ **Question 7** On peut faire plus efficace (dans certains cas), mais le plus simple est :

```
let rec elimine_constantes = function
  | Et (f, g) ->
    begin match elimine_constantes f, elimine_constantes g with
      | C false, _ | _, C false -> C false
      | C true, h | h, C true -> h
      | f', g' -> Et (f', g')
    end
  | Ou (f, g) ->
    begin match elimine_constantes f, elimine_constantes g with
      | C true, _ | _, C true -> C true
      | C false, h | h, C false -> h
      | f', g' -> Ou (f', g')
    end
  | Imp (f, g) ->
    begin match elimine_constantes f, elimine_constantes g with
      | C false, _ | _, C true -> C true
      | C true, h -> h
      | f', C false -> Non f'
      | f', g' -> Imp (f', g')
    end
  | Non f ->
    begin match elimine_constantes f with
      | C b -> C (not b)
      | f' -> Non f'
    end
  | f -> f
```

▶ **Question 8**

```
let rec substitue f i g =
  match f with
  | V j when i = j -> g
  | Non f' -> Non (substitue f' i g)
  | Et (f1, f2) -> Et (substitue f1 i g, substitue f2 i g)
  | Ou (f1, f2) -> Ou (substitue f1 i g, substitue f2 i g)
  | Imp (f1, f2) -> Imp (substitue f1 i g, substitue f2 i g)
  | _ -> f
```

▶ **Question 9** On a $s(f[\top/x_0]) = (x_1 \wedge x_2) \wedge \neg x_2$ et $s(f[\bot/x_0]) = \top$. On peut remarquer que $s(f[\top/x_0]) \equiv \bot$, mais que cette simplification ne sera pas faite.

▶ **Question 10** On montre par induction structurelle sur f que si f ne contient pas de variable, alors $s(f)$ est de la forme `C b`.

- Le seul cas de base à considérer est $f =$ `C b`, on a alors $s(f) =$ `C b`.
- Si $f = \mathrm{Et}(f_1, f_2)$, alors f_1 et f_2 ne contiennent pas de variable, donc $s(f_1)$ et $s(f_2)$ sont réduits à une feuille `C b`. La lecture du code montre alors que $s(f)$ est également une feuille `C b`.
- Les cas `Et`, `Imp` et `Non` se traitent de manière similaire.

Le nombre de variables (distinctes) présentes dans la formule diminue strictement à chaque appel récursif. Toute chaîne d'appel finit donc par arriver sur une formule sans variable, qui est un cas de base ($s(f)$ réduite à une constante) d'après ce qui précède.

▶ **Question 11** La formule est satisfiable si l'arbre associé possède au moins une feuille étiquetée `true`, une tautologie si toutes ses feuilles sont étiquetées `true`. Notons que :

- l'arbre obtenu dépend du choix de la variable à chaque étape, mais cette propriété est vraie quel que soit l'arbre ;
- une démonstration n'est pas difficile à rédiger ; il faut utiliser le fait que, pour toute variable x, $f \equiv (x \wedge f[\top/x]) \vee (\neg x \wedge f[\bot/x])$.

▶ **Question 12** On traduit exactement l'énoncé :

```
let rec min_var = function
  | C _ -> max_int
  | V i -> i
  | Et (f, g) | Ou (f, g) | Imp (f, g) ->
    min (min_var f) (min_var g)
  | Non f -> min_var f

let rec construire_arbre formule =
  match elimine_constantes formule with
  | C b -> Feuille b
  | f ->
    let i = min_var f in
    let f_bot = substitue f i (C false) in
    let f_top = substitue f i (C true) in
    Noeud (i, construire_arbre f_bot, construire_arbre f_top)
```

▶ **Question 13** On cherche une feuille `true` :

```
let rec satisfiable_via_arbre f =
  let rec aux = function
  | Feuille b -> b
  | Noeud (_, g, d) -> aux g || aux d in
  aux (construire_arbre f)
```

▶ **Question 14** On peut par exemple définir pour chaque sommet i :

- $A_i = \bigvee_{c=0}^{k-1} x_{i,c}$ (le sommet a au moins une couleur) ;
- pour chaque couleur c, une formule $B_{i,c}$ qui traduit le fait que si i est colorié avec c, alors il n'est pas colorié avec une autre couleur, et ses voisins ne sont pas coloriés avec c :

$$B_{i,c} = x_{i,c} \to \neg \left(\bigvee_{c' \neq c} x_{i,c'} \vee \bigvee_{\{i,j\} \in E} x_{j,c} \right)$$

- les contraintes pour i s'écrivent alors $A_i \wedge \bigwedge_{c=0}^{k-1} B_{i,c}$;
- la formule cherchée est $\bigwedge_{i=0}^{n-1} A_i$.

▶ **Question 15** Il faut s'organiser un peu si l'on veut éviter que le code ne devienne trop compliqué.

```
let range n = List.init n (fun i -> i)

let rec binarise_ou formules =
  match formules with
  | [] -> C false
  | [f] -> f (* pas indispensable *)
  | f :: fs -> Ou (f, binarise_ou fs)

let rec binarise_et formules =
  match formules with
  | [] -> C true
  | [f] -> f (* pas indispensable *)
  | f :: fs -> Et (f, binarise_et fs)

let encode g k =
  let n = Array.length g in
  let var i c = V (i * k + c) in
  let est_colorie i =
    binarise_ou (List.init k (fun c -> var i c)) in
  let contraintes i =
    let contraintes_couleur c =
      let voisins = List.map (fun j -> var j c) g.(i) in
      let autres_couleurs = List.filter (fun x -> x <> c) (range k) in
      let unique = List.map (fun c' -> var i c') autres_couleurs in
      Imp (var i c, Non (binarise_ou (voisins @ unique))) in
    Et (est_colorie i, binarise_et (List.init k contraintes_couleur)) in
  binarise_et (List.init n contraintes)
```

▶ **Question 16**

```
let est_k_coloriable test_satisfiable g k =
  let formule = encode g k in
  Printf.printf "Test %d-coloriable\n%!" k;
  Printf.printf "Nombre de variables : %d\n%!" (1 + var_max formule);
  Printf.printf "Taille : %d\n%!" (taille formule);
  test_satisfiable formule
```

▶ **Question 17** On peut déjà remarquer que $\chi(G)$ est inférieur à $\Delta(G)+1$, où $\Delta(G)$ est le degré maximum des nœuds de G.Pour limiter le nombre d'appels à `est_k_coloriable`, on procède ensuite par dichotomie :

```
let degre_max g =
  let rec aux i =
    if i = Array.length g then 0
    else max (List.length g.(i)) (aux (i + 1)) in
  aux 0
```

```
let chromatique test_satisfiable g =
  let rec aux i j =
    if i = j then begin
      Printf.printf "\nchi(G) = %d\n%!" i;
      i
    end else begin
      Printf.printf "\n%d <= chi(G) <= %d\n%!" i j;
      let m = (i + j) / 2 in
      if est_k_coloriable test_satisfiable g m then aux i m
      else aux (m + 1) j
    end in
  let n = Array.length g in
  let dmax = degre_max g in
  let p = (Array.fold_left (fun acc u -> acc + List.length u) 0 g) / 2 in
  Printf.printf "%d sommets, %d arêtes, degré max %d\n%!" n p dmax;
  aux 1 (degre_max g + 1)
```

On a ajouté quelques affichages pour permettre de juger de l'avancement des calculs (qui peuvent être très longs !).

► **Question 18**

```
let lire_dimacs filename =
  let ic = open_in filename in
  let rec get_next () =
    let s = input_line ic in
    if s.[0] = 'c' then get_next ()
    else s in
  let n, _ = Scanf.sscanf (get_next ()) "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
          (get_next ())
          "e %d %d"
          (fun i j -> (i - 1, j - 1)) in
      if not (List.mem j g.(i)) then g.(i) <- j :: g.(i);
      if not (List.mem i g.(j)) then g.(j) <- i :: g.(j)
    done;
    assert false
  with
  | End_of_file -> g
```

► **Question 19** On doit trouver $\chi(G) = 4$ (rapidement, si tout se passe bien).

► **Question 20** Une première possibilité naturelle :

```
let rec simplifie x b = function
  | V i when i = x -> C b
  | Et (f, g) ->
    begin match simplifie x b f, simplifie x b g with
      | C true, h | h, C true -> h
      | C false, _ | _, C false -> C false
      | f', g' -> Et (f', g')
    end
  | Ou (f, g) ->
    begin match simplifie x b f, simplifie x b g with
      | C true, _ | _, C true -> C true
      | C false, h | h, C false -> h
      | f', g' -> Ou (f', g')
    end
  | Imp (f, g) ->
    begin match simplifie x b f, simplifie x b g with
      | C false, _ | _, C true -> C true
      | C true, g' -> g'
      | f', C false -> Non f'
      | f', g' -> Imp (f', g')
    end
  | Non f ->
    begin match simplifie x b f with
      | Non f' -> f'
      | C b' -> C (not b')
      | f' -> Non f'
    end
  | f -> f
```

Cette version permet d'être bien plus efficace qu'en construisant explicitement l'arbre, mais on peut en fait faire mieux. En effet, on fait systématiquement les appels sur les deux sous-arbres alors que dans certains cas (`Et (C false, g)`, par exemple), on aurait pu s'en passer. On préfèrera donc le code suivant (un peu lourd, certes) :

```
let rec simplifie2 x b = function
  | V i when i = x -> C b
  | Et (f, g) ->
    begin match simplifie2 x b f with
      | C false -> C false
      | C true -> simplifie2 x b g
      | f' -> begin match simplifie2 x b g with
          | C false -> C false
          | C true -> f'
          | g' -> Et (f', g')
        end
    end
  | Ou (f, g) ->
    begin match simplifie2 x b f with
      | C true -> C true
      | C false -> simplifie2 x b g
      | f' -> begin match simplifie2 x b g with
          | C true -> C true
          | C false -> f'
          | g' -> Ou (f', g')
        end
    end
  | Imp (f, g) ->
    begin match simplifie2 x b f with
      | C false -> C true
      | C true -> simplifie2 x b g
      | f' -> begin match simplifie2 x b g with
          | C true -> C true
          | C false -> Non f'
          | g' -> Imp (f', g')
        end
    end
  | Non f ->
    begin match simplifie2 x b f with
      | Non f' -> f'
      | C b' -> C (not b')
      | f' -> Non f'
    end
  | f -> f
```

Bien choisir la variable à utiliser a chaque étape a un impact majeur sur l'efficacité de la méthode, mais dans notre cas (aucune contrainte sur la forme de la formule) il est difficile de trouver une bonne heuristique. Nous allons donc prendre la première variable qui nous tombera sous la main :

```
let rec premiere_var = function
  | V i -> Some i
  | C _ -> None
  | Non f -> premiere_var f
  | Et (f, g) | Ou (f, g) | Imp (f, g) ->
    begin match premiere_var f with
      | Some i -> Some i
      | None -> premiere_var g
    end


let satisfiable formule =
  let rec aux f =
    match f, premiere_var f with
    | C b, None -> b
    | _, None -> failwith "impossible"
    | _, Some i ->
      aux (simplifie2 i true f) || aux (simplifie2 i false f) in
  aux (elimine_constantes formule)
```

► **Question 21** Sur ma machine, on obtient :

```
$ time ./quine3 myciel4.col
23 sommets, 71 arêtes, degré max 11

1 <= chi(G) <= 12
Test 6-coloriable
Nombre de variables : 138
Taille : 4143

1 <= chi(G) <= 6
Test 3-coloriable
Nombre de variables : 69
Taille : 1704

4 <= chi(G) <= 6
Test 5-coloriable
Nombre de variables : 115
Taille : 3238

4 <= chi(G) <= 5
Test 4-coloriable
Nombre de variables : 92
Taille : 2425

chi(G) = 5
Le nombre chromatique vaut 5.
./quine3 myciel4.col  0,93s user 0,01s system 99% cpu 0,939 total
```