TP 18 : Logique propositionnelle, algorithme de Quine

Exercice 1 – Logique propositionnelle

▶ **Question 1** Proposez un type `formule` des formules propositionnelles, comprenant la possibilité d'écrire les formules $\top$, $\bot$, et indexant les variables propositionnelles par des entiers naturels.

On utilisera le type `type valuation = bool array` pour représenter une valuation sur les variables propositionnelles $p_0, \dots, p_{n-1}$, avec n la taille du tableau. Ainsi, pour $n = 4$, la valuation `[|true; false; true; false|]` représente la valuation ν telle que $\nu(p_0) = \nu(p_2) =$ vrai et $\nu(p_1) = \nu(p_3) =$ faux.

▶ **Question 2** Soit $\varphi = (\bot \vee p_0) \to (\neg p_1 \wedge p_2 \vee p_0) \wedge (\top \to p_2 \vee p_3)$. Écrivez-la dans une variable `exemple : formule`. Déterminer sa taille et sa hauteur.

▶ **Question 3** Écrire une fonction `taille : formule -> int` qui retourne la taille de la formule en entrée, c'est-à-dire le nombre de nœuds de l'arbre syntaxique associé.

▶ **Question 4** Écrire une fonction `ind_var_max : formule -> int` qui calcule l'indice maximal des variables propositionnelles dans la formule en entrée. La fonction renverra -1 si la formule en entrée n'a pas de variable propositionnelle.

▶ **Question 5** Écrire une fonction `valeur_verite : formule -> valuation -> bool` qui, pour une formule φ et une valuation ν des variables propositionnelles de la formule, renvoie si $\nu \models \varphi$.

▶ **Question 6** Déterminer sa complexité en fonction de la taille de la formule en entrée.

Exercice 2 – Satisfaisabilité par force brute

On peut aussi voir une valuation comme un nombre binaire (positif), en interprétant `true` comme 1 et `false` comme 0. Ainsi, à une valuation ν sur $\mathcal{P} = [\![0, n-1]\!]$, on associe l'entier $x = \sum_{k=0}^{n-1} \nu(k) 2^{n-1-k}$.
On peut donc itérer sur les valuations en les incrémentant en tant que nombre binaire positif : la première valuation sera `[|false; false; ...|]`, la suivante `[|false; ...; false; true]`, et la dernière `[|true; true; ...; true|]`.
On utilise dans cette partie une exception, utilisée comme suis :

OCaml
```ocaml
(* déclaration d'une nouvelle exception sans arguments *)
exception Debordement
(* on peut aussi faire des exceptions avec [exception E of int * int] *)

(* lève l'exception Debordement *)
let () = raise Debordement
```

▶ **Question 1** Écrire une fonction `incrementer_valuation : valuation -> unit` qui transforme *en place* la valuation en entrée en une valuation correspondant au nombre $x+1$. S'il n'est pas possible de l'incrémenter, on lève l'exception `Debordement`.

▶ **Question 2** En déduire une fonction `satisfaisable_brute : formule -> bool` qui résout le problème SAT.

▶ **Question 3** De la même façon, écrire une fonction `equivalent : formule -> formule -> bool`

qui vérifie que les deux formules en entrée sont équivalentes.

▶ **Question 4** Écrire la fonction `supprimer_impl : formule -> formule` qui enlève les implications de la formule grâce à l'équivalence $\varphi \to \psi \equiv \neg\varphi \vee \psi$ et la fonction `repousser_neg : formule -> formule` qui repousse les négations vers les variables propositionnelles.

★ **Question 5** Écrire une fonction `fnc : formule -> formule` qui calcule la forme normale conjonctive de la formule en entrée. La tester sur l'exemple.

Exercice 3 – Algorithme de Quine

Pour une formule de la logique propositionnelle, on peut toujours la *simplifier* pour obtenir une formule équivalente égale à $\top$, $\bot$ ou ne contenant ni l'une ni l'autre. Pour cela, il suffit d'utiliser les équivalences suivantes :

Définition 1 – Règles de simplification de Quine

$$\neg\top \equiv \bot \quad \varphi \vee \top \equiv \top \quad \varphi \vee \bot \equiv \varphi \quad \top \wedge \varphi \equiv \varphi \quad \bot \wedge \varphi \equiv \bot$$
$$\neg\top \equiv \bot \quad \top \vee \varphi \equiv \top \quad \bot \vee \varphi \equiv \varphi \quad \varphi \wedge \top \equiv \varphi \quad \varphi \wedge \bot \equiv \bot$$

▶ **Question 1** Compléter ces règles de simplification afin d'autoriser la simplification de $\to$.

▶ **Question 2** En utilisant la définition 1 en une fonction `sub_et_simp : formule -> int -> formule -> formule` telle que `substitution phi k psi` est évalué en une formule équivalente $\varphi\{p_k \mapsto \psi\}$ qui est soit égale à $\top$ ou $\bot$, soit n'en contient aucun des deux. *On suppose que φ ne contient ni $\top$ ni $\bot$, et que $\psi = \top$ ou $\bot$. La complexité de cette fonction doit être linéaire en la hauteur de φ.*

▶ **Question 3** Appliquer la substitution à `exemple`, p_0 et la formule $p_1 \vee \neg p_2$.

L'idée de l'algorithme de Quine est la suivante :

1. on choisit une variable propositionnelle p_i,
2. on lui affecte une valeur de vérité (disons `vrai`), en calculant $\varphi\{p_i \mapsto \top\}$;
3. on simplifie la formule obtenue avec les règles de la Définition 1,
4. on applique récursivement l'algorithme sur la formule obtenue (s'il reste une variable propositionnelle à fixer). Si cela ne donne rien, on revient en arrière pour choisir l'autre valeur de vérité

Les étapes 1 à 4 décrivent un algorithme de retour sur trace pour déterminer la satisfaisabilité d'une formule de la logique propositionnelle. Dans ce TP, on va plutôt construire explicitement l'*arbre de décision* associé à cet algorithme : un arbre dans lequel chaque nœud étiqueté par k modélise le choix de la valeur de vérité de p_k : le fils gauche correspond au cas $\nu_{p_k} =$ `vrai`, et le fils droit le cas $\nu_{p_k} =$ `faux`. Pour l'implémenter, on va y utiliser une stratégie récursive :

Pseudo-code

```
1 : Fonction Quine(φ)
2 :     ψ ← Simpl(φ)
3 :     Si ψ ∉ {⊤, ⊥} alors
4 :         Soit p ∈ P présente dans ψ,        ▷ qui existe, sinon ψ serait ⊤ ou ⊥
5 :         retourne l'arbre suivant :
              Noeud(p, Quine(ψ{p ↦ ⊤}), Quine(ψ{p ↦ ⊥}))
6 :     Sinon
7 :         retourne Feuille(ψ)                ▷ ψ ∈ {⊤, ⊥}
```

Maintenant, on a tout ce qu'il faut pour se lancer dans l'implémentation de l'algorithme de Quine, en utilisant les règles de la Définition 1. Pour représenter un arbre de décision, on va utiliser un type d'arbres binaires stricts, dont les feuilles sont étiquetées par des booléens.

OCaml
```
type arbre_decision =
  | Feuille of bool
  | Noeud of arbre_decision * int * arbre_decision
```

▶ **Question 4** Écrire une fonction `construire_arbre_decision : formule -> arbre_decision`. On utilisera la fonction `ind_var_max` pour choisir la variable propositionnelle à substituer à chaque appel récursif.

▶ **Question 5** En choisissant toujours la proposition de plus grand index dans la formule d'entrée, proposer une suite de formules $\varphi_0, \varphi_1, \dots$ telle que l'arbre de décision calculé est de taille exponentielle en la taille de la formule.

▶ **Question 6** Écrire une fonction `satisfaisable_quine : formule -> bool` déterminant si une formule est satisfaisable.

▶ **Question 7** Déterminer sa complexité temporelle et spatiale.

★ **Question 8** Proposer une heuristique alternative `choix_proposition : formule -> int` qui choisit une variable propositionnelle à substituer « plus intelligemment ». Montrer que la complexité temporelle n'est pas meilleure dans le pire cas.

▶ **Question 9** Écrire une fonction `valide : formule -> bool` déterminant si une formule est valide.

★ **Question 10** Écrire une fonction `modele_opt : formule -> valuation option` qui renvoie un modèle de l'entrée (`None` s'il n'en existe pas).

★ **Question 11** Écrire une fonction `nombre_modeles : formule -> int` dénombrant les modèles d'une formule. On supposera que l'ensemble des variables propositionnelles est $\{p_0, \dots, p_k\}$ avec p_k la variable propositionnelle d'indice le plus grand apparaissant dans la formule en entrée, et que $k < 62$ (★ pourquoi ?).

★ **Question 12** Appliquer la fonction `construire_arbre_decision` à la formule suivante : $(p \wedge q) \vee \neg p \vee \neg q$. Que remarque-t-on ? Écrire une fonction `arbre_opti : formule -> arbre_decision` qui évite ce problème.

Exercice 4 – Interpréteur MiniSAT

MiniSAT est un SAT-solver très efficace, basé sur l'algorithme DPLL. On va ici écrire un interpréteur pour MiniSAT, qui prend en entrée un fichier de description d'une formule en CNF et renvoie si elle est satisfaisable ou non.
C'est un programme en ligne de commande, qui prend en argument le nom du fichier contenant la formule en CNF dont le format est comme suis :

Shell

```
$ cat exemple.cnf
c Ligne de commentaire
c Autre ligne de commentaire
p cnf 3 5
1 -2 3 0
2 3 0
-1 -2 -3 0
1 -3 0
1 2 0
$ ./minisat exemple.cnf
================================[MINISAT]================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
=========================================================================
|         0 |       5       12 |       1       0        0     nan |  0.000 % |
=========================================================================
restarts              : 1
conflicts             : 1              (inf /sec)
decisions             : 3              (inf /sec)
propagations          : 5              (inf /sec)
conflict literals     : 1              (0.00 % deleted)
Memory used           : 1.69 MB
CPU time              : 0 s

SATISFIABLE
```

— Une ligne commençant par un c est un commentaire et doit être ignorée. On supposera pour simplifier que ces lignes sont nécessairement au début du fichier.
— La ligne `p cnf 3 5` signifie que le problème contient 3 variables et 5 clauses. Les variables sont numérotées à partir de 1 (donc de 1 à 3 dans l'exemple ci-dessus), ce qui permet de représenter le littéral x_i par l'entier i et $\neg x_i$ par $-i$.
— Il y a ensuite une clause par ligne : les différents littéraux apparaissant dans la clause sont séparés par des espaces, et il y un 0 à la fin pour signaler la fin de la clause.
— On pourra supposer qu'une clause ne contient pas deux fois le même littéral, ni un littéral et sa négation.

On pourra utiliser la fonction suivante pour lire une clause dans un fichier :

OCaml

```
let string_to_int_list s =
  s |> String.trim |> String.split_on_char ' ' |> List.map int_of_string
```

▸ **Question 1** Écrire une fonction `parse_cnf : string -> formule` qui prend en entrée le nom d'un fichier de description d'une formule en CNF et renvoie la formule correspondante.

▸ **Question 2** Écrire une fonction `minisat : string -> unit` qui prend en entrée le nom du fichier de description d'une formule en CNF et renvoie si elle est satisfaisable ou non.

▸ **Question 3** Tester votre programme sur un fichier de description d'une formule en CNF plus grosse, comme celles disponibles en bas de cette page : `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`. En utilisant l'utilitaire `time`, comparer les trois méthodes de résolution de la

satisfaisabilité de formules de logique propositionnelle implémentées dans ce TP (force brute, algorithme de Quine, MiniSAT).