Programmation Python
TP n° 9 : Formes normales et satisfiabilité

L'objectif de ce TP est de mettre en œuvre plusieurs algorithmes de formes normales pour la logique propositionnelle, comme vu dans le cours de Logique, et de les utiliser pour convertir des problèmes de satisfiabilité en un format lisible par un solveur SAT.
Référence : [1] S. Schmitz, Introduction à la logique.

I) Formules, affectations, équivalence

Rappelons qu'une *formule* de logique propositionnelle est définie recursivement comme une variable propositionnelle p_i, où i est un entier positif, une négation $\neg\phi$, une conjonction $\phi \wedge \psi$ ou une disjonction $\phi \vee \psi$, où ϕ et ψ sont des formules. On admet $\phi \to \psi$ comme une abréviation de $\neg\phi \vee \psi$ et $\bot$ comme une abréviation de $p_1 \wedge \neg p_1$.

Question 1. Définissez une classe `Formule` avec quatre sous-classes `Var`, `Non`, `Et`, et `Ou`, avec des constructeurs et des méthodes `__str__`. Les constructeurs de `Et` et `Ou` accepteront un nombre arbitraire d'arguments.

On modélisera une *affectation* comme une liste de valeurs booléennes. Étant donnée une affectation de longueur m, toute formule ϕ telle que toutes les propositions dans ϕ ont un indice $\leqslant m$ peut être évaluée.

Question 2. Écrivez une fonction `max_var` qui prend une formule et retourne l'indice maximale des variables propositionnelles dans la formule.

Question 3. Écrivez une fonction `valeur_de_verite` qui prend une formule et une affectation, et retourne la valeur de vérité de la formule si possible, ou lève une exception sinon.

On rappelle que deux formules ϕ et ψ sont (semantiquement) *équivalentes* si, pour toute affectation possible des variables, les valeurs de vérités de ϕ et ψ coïncident.

Question 4. Écrivez une fonction `is_equiv` qui prend deux formules `f` et `g` et qui retourne `True` si `f` et `g` sont équivalentes, et `False` sinon. Quelle est la complexité en temps de la fonction ? Est-ce optimale ?

II) Forme normale négative, conjonctive et clausale

Rappelons qu'une formule est en *forme normale négative* si les seules négations sont appliquées à des variables propositionnelles. Toute formule a une forme normale négative équivalente [1, Def. 7.1].

Question 5. Définissez une fonction `nnf` qui prend une formule en entrée et retourne une formule équivalente en forme normale négative. Testez votre fonction sur la loi de Pierce $((P \to Q) \to P) \to P$ [1, Ex. 7.2].

Rappelons qu'une formule est en *forme normale conjonctive* (cnf) si elle est en forme normale négative, et qu'aucune conjonction n'apparaît sous la portée d'une disjonction. Un *littéral* est une formule de la forme p_i ou $\neg p_i$.

Question 6. Écrivez des fonctions `is_literal` et `is_cnf` qui vérifient si une formule donnée est un littéral ou est en cnf, respectivement.

Une *clause* est une disjonction de littéraux. Une formule est en cnf si, et seulement si, elle est une conjonction de clauses. Pour k un nombre naturel et ϕ une formule en cnf, on dit que ϕ est en k-cnf si chaque clause de ϕ contient au plus k littéraux.

Question 7. Écrivez une fonction `width` qui prend une formule en cnf et retourne le plus petit k tel que ϕ est en k-cnf.

Toute formule a une forme normale conjonctive équivalente [1, Sec. 7.2.1].

Question 8. Définissez une fonction `cnf` qui prend en entrée une formule et retourne une formule équivalente en forme normale conjonctive.

On peut associer à toute clause C une liste d'entiers, de la manière suivante. D'abord, à une proposition p_i, on associe l'entier positif i, et à une proposition négative $\neg p_i$, on associe l'entier négatif $-i$. On associe alors à C la liste d'entiers associée aux littéraux qui y figurent. Par exemple, $p_1 \vee \neg p_2 \vee p_1$ est associé à la liste `[1,-2,1]`.

Question 9. Définissez une fonction `list_of_clause` qui prend une clause en entrée et retourne la liste d'entiers correspondante.

Une formule ϕ en cnf a naturellement associée une liste L de listes d'entiers : les éléments de la liste L sont les listes associées aux clauses de ϕ.

Question 10. Définissez une fonction `list_list_of_cnf` qui prend une formule en cnf en entrée et retourne la liste de listes d'entiers correspondante.

Rappelons [1, Sec. 8.2.1] que le format DIMACS est un format standard de fichier pour représenter des formules en cnf. La première ligne du fichier (sauf lignes commentaires) a la forme `p cnf x y` où `x` est le nombre de propositions utilisées, et `y` est le nombre de clauses. Après cette première ligne, chaque ligne suivante représente une clause, comme une séquence d'entiers séparés par des espaces, terminée par le nombre `0` comme marqueur de fin de ligne.

Question 11. Écrivez une fonction qui prend une formule en forme normale conjonctive et retourne une chaîne de caractère au format DIMACS qui représente la formule.

Question 12. On définit la longueur d'une formule comme le nombre de symboles dans la formule. Soit $A\colon \mathbb{N} \to \mathbb{N}$ la fonction définie par : $A(n)$ est la longueur maximale des cnf de formules ϕ de longueur $\leqslant n$. Démontrez que $A(n) \geqslant 2^{\frac{n}{2}}$ pour tout $n \geqslant 1$.

III) Forme de Tseitin

Rappelons que ϕ est équi-satisfiable avec ψ lorsque ϕ est satisfiable si, et seulement si, ψ est satisfiable. On vise maintenant à donner une procédure en temps linéaire pour produire une formule *équi-satisfiable* en 3-cnf pour toute formule donnée [1, Sec. 7.2.2].
Pour ce faire, on veut considérer $\vee$ et $\wedge$ comme des opérations *binaires*. On dit qu'une formule ϕ est en forme *binaire* si toute occurrence de $\vee$ ou $\wedge$ prend au plus 2 arguments. Par exemple, la formule `Or(Var(1),Var(2),Var(3),Var(4),Var(5))` n'est pas en forme binaire, mais la formule `Or(Or(Var(1),Var(2)),Or(Or(Var(3),Var(4)),Var(5)))` l'est.

Question 13. Écrivez une fonction `binaryform` qui prend une formule `f` et qui retourne une formule équivalente en forme binaire.

Question 14. Écrivez une fonction `subformulas` qui prend une formule `f` et retourne la liste des sous-formules de `f`. Donnez une estimation de la complexité dans le pire cas de cette fonction, en temps et en espace.

Soit ϕ une formule en forme normale négative et binaire. Soit m l'indice maximal des propositions figurant dans ϕ. Soit S l'ensemble des sous-formules de ϕ, ordonné de manière arbitraire, écrivons $S = \{\phi_1, \ldots, \phi_s\}$ et supposons $\phi_1 = \phi$. Pour chaque $1 \leqslant i \leqslant s$, écrivons $q_i := p_{m+i}$.
On définit maintenant, pour chaque $1 \leqslant i \leqslant s$, une formule ψ_i comme suit. Si $\phi_i = \phi_{j_1} \vee \phi_{j_2}$, on pose $\psi_i = q_{j_1} \vee q_{j_2}$ et, de manière similaire, si $\phi_i = \phi_{j_1} \wedge \phi_{j_2}$, on pose $\psi_i = q_{j_1} \wedge q_{j_2}$. Enfin, si ϕ_i est un littéral, on pose $\psi_i = \phi_i$.
La *forme de Tseitin* de ϕ est la formule $T(\phi) := q_1 \wedge \bigwedge_{i=1}^{s}(q_i \to \psi_i)$.

Question 15. Donnez une formule en 3-cnf équivalente à $T(\phi)$.

Question 16. Écrivez une fonction `tseitin` qui prend en entrée une formule ϕ en forme normale négative, et retourne la forme de Tseitin de ϕ en 3-cnf. Évaluez sa complexité en temps et en espace.

Question 17. Prouvez que la forme de Tseitin de ϕ est équi-satisfiable avec ϕ.

Question 18. Écrivez une fonction `extend_affectation` qui prend en entrée une affectation satisfaisante de ϕ et retourne une affectation satisfaisante de $T(\phi)$.

IV) Application

Sur un échiquier de dimension $n \times n$, la reine peut se déplacer en un coup d'un nombre arbitraire de cases, en diagonale, horizontale ou verticale. Une reine *attaque* une autre pièce sur l'échiquier si elle peut atteindre la case de cette pièce au prochain coup. Le *problème des n reines* demande s'il est possible de placer n reines sur un échiquier de taille $n \times n$ de telle manière qu'aucune des reines n'attaque une autre reine de l'échiquier.

Question 19. Modélisez le problème des n reines comme un problème de satisfiabilité en logique propositionnelle : écrivez une fonction `queens(n)` qui produit une formule ϕ dont les affectations satisfaisantes sont en bijection avec les solutions du problème des n reines. Pour cela, introduisez n^2 propositions, chacune représentant la présence d'une reine sur la case correspondante de l'échiquier.

Question 20. Utilisez votre programme pour créer un fichier au format DIMACS représentant le problème des n reines. Testez ceci avec un solveur SAT (par exemple MiniSAT). Écrivez une fonction qui affiche la solution sur un échiquier.

Extensions possibles :

— écrire des fonctions permettant de transformer un puzzle de Sudoku ou un puzzle de StarBattle en un problème DIMACS ;

— écrire un parseur de formules et/ou de fichiers DIMACS ;

— représenter des formules de premier ordre : ajouter des quantificateurs, notions sémantiques, forme prénexe.