Projet de TP n° 2:

Traitement d'ensembles de clauses et introduction à la démonstration automatique basée sur l'algorithme de Davis et Putnam

DLST L1
Université Grenoble Alpes

TP, projet 2 (rév. SP 10/4/2019)
INF202 2018/2019

Objectifs

L'objectif du projet est de comprendre comment on peut faire de la démonstration automatique avec un ordinateur. On prend le cas particulier, vu en cours, où on veut montrer qu'une argumentation du type $\mathcal{H} \vDash C$ est valide ou non.

On sait (cf. cours) qu'une telle argumentation est valide **ssi** (théorème de l'absurde) $\mathcal{H} \cup \{C\} \vDash$ `faux`, c'est à dire si $\mathcal{H} \cup \{C\}$ est contradictoire.

Un autre problème est de déterminer si un ensemble $\mathcal{E}$ de formules propositionnelles est satisfaisable, c'est à dire s'il existe une *réalisation* v telle que $v(\mathcal{E}) = 1$: v rend vraie chaque formule de $\mathcal{E}$ $((\forall A \in \mathcal{E})\ [v(A) = 1])$, et si oui de fournir au moins une réalisation de $\mathcal{E}$.

Or on sait que tout ensemble $\mathcal{E}$ de formules propositionnelles peut être transformé en un ensemble $\mathcal{F}$ de clauses équivalent, c'est à dire qui ont la même valeur de vérité, pour toute valuation v (fonction des atomes propositionnels dans $\mathbf{2} = \{0, 1\}$, 0 étant mis pour `faux` et 1 pour `vrai`).

Le but de ce TP est d'apprendre comment on peut représenter en Python un ensemble de clauses, comment on peut le lire et l'écrire, comment on peut implémenter une ou deux versions de l'algorithme de Davis et Putnam, et comment on peut utiliser les programmes construits à l'occasion de ce TP pour résoudre quelques uns des exercices de TD.

Les tâches à réaliser sont dans les sections 2 et 3.

Ce qu'il y a à rendre et comment le rendre est décrit ci-dessous.

Rendu final

Vous devrez rendre plusieurs fichiers dans une archive zip, à envoyer à votre référent de TP, avant le **3 Mai 2019, 12h heure de Paris**. Le fichier sera nommée INF202-TP-projet2-<votreNomDeFamille>-<date>.zip (ex : INF202-TP-projet2-Dupont-20190429.zip).

Fichiers :

- au moins 3 fichiers .txt contenant un ensemble de clauses en convention d'entrée (S 2.1)
- un fichier .py contenant toutes les procédures .py, chacune suivie d'un petit "test unitaire" (S 2.2, 2.3, 2.4 et 3)
- plusieurs fichiers .log contenant les traces d'exécution pour vos jeux d'essai. (S 3)
- un rapport dans un fichier LISEZ-MOI.txt, présentant brièvement la répartition du travail fait, en particulier l'organisation du travail (qu'il s'agisse d'un binôme ou pas), la répartition du temps entre les différentes tâches, et une auto-évaluation.

Mail des référents
— joanny.perret@grenoble-inp.fr
— jose-ignacio.requeno-jarabo@univ-grenoble-alpes.fr
— mouhcine.mendil@inria.fr
— sergi.pujades-rocamora@univ-grenoble-alpes.fr

1 Représentations d'un ensemble de clauses (en lecture, traitement et écriture)

1.1 Représentation d'entrée

On veut permettre d'écrire n'importe quelle clause, avec ses littéraux dans n'importe quel ordre, et en permettant les clauses tautologiques (qui contiennent au moins une paire de littéraux conjugués).

On adopte les conventions suivantes :

— Les atomes seront pris dans l'ensemble $\{a, b, c \ldots, z\}$ et auront les numéros $\{1, 2, 3 \ldots, 26\}$. Nous convenons donc que $p_1 = a, p_2 = b, \ldots, p_{26} = z$.

— La représentation externe (pour la lecture) d'un littéral sera son numéro, positif ou négatif.
Par exemple, pour noter $\neg c$ en entrée, on écrira -3.

— On représentera en entrée une clause par la suite des valeurs (internes) de ses littéraux, et on l'encadrera par [...].
Par exemple, la clause

$$c \vee g \vee \neg b \vee \neg e \vee \neg f$$

(que nous notons souvent $cg\bar{b}\bar{e}\bar{f}$ pour abréger, et pour faciliter les calculs manuels) sera représentée par :

```
[ 3, 7, -2, -5, -6 ]
```

— Pour pouvoir lire facilement (en Python) un ensemble de clauses, on l'encadrera également par [...]. Donc $\{cg\bar{b}\bar{e}\bar{f}, aeg\bar{c}\bar{d}\bar{e}\}$ pourra être entré comme :

```
[   [  3,  7,  -2,  -5,  -6       ]
    [  1,  5,  7,   -3,  -4,  -5  ]  ]
```

On appellera cette forme d'écriture `convEnt1`.

— On définit aussi la convention `convEnt2`, plus naturelle mais un peu plus difficile à faire lire par un programme : on utilise les atomes tels quels, et on note la négation par "-" devant un atome.
Par exemple, pour l'ensemble des deux clauses précédentes :

```
[   [  c,  g,  -b,  -e,  -f       ]
    [  a,  e,  g,   -c,  -d,  -e  ]  ]
```

1.2 Représentation interne

Pour opérer sur des ensembles de clauses, il est plus commode d'adopter une représentation interne matricielle, à 26 colonnes (une par atome), et n lignes si on a n clauses.

La clause i sera représentée par la ligne i, et cette ligne aura donc 26 éléments. Soit M une telle matrice, on mettra dans la case $M[i][j]$:

-1 si le littéral $\overline{p_j}$ apparaît dans la clause, et pas p_j.
0 si l'atome p_j n'apparaît pas dans la clause (donc, ni p_j ni $\overline{p_j}$).
1 si le littéral p_j apparaît dans la clause, et pas $\overline{p_j}$.
2 si le littéral p_j et le littéral $\overline{p_j}$ apparaissent dans la clause.

Avec l'exemple précédent, cela donne

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	...	z
0	-1	1	0	-1	-1	1	0	0	0	0	0	0	0	0	...	0
1	0	-1	-1	2	0	1	0	0	0	0	0	0	0	0	...	0

1.3 Représentations de sortie

On définit 3 conventions de sortie pour imprimer un ensemble de clauses (on utilise un exemple plus petit que le précédent) :

```
0: formeEtOu:                (a or -b or d) and (-a or c or -e)
1: tableauPy (à la Python):  [ [a -b d] [-a c -e] ]
2: sommeMonomes:             a-bd + -ac-e
```

2 Préparation, lecture et écriture d'ensembles de clauses

2.1 Préparation d'un jeu de test

Préparer au moins 3 fichiers contenant chacun un ensemble de clauses, en convention `convEnt1` et `convEnt2`, provenant d'exemples tirés du cours ou des TD.

2.2 Lecture

Écrire un programme `ensClauses = lireEnsClauses(ficEntree, convEnt)` qui produit dans `ensClauses` la représentation interne de l'ensemble de clauses contenu dans `ficEntree` selon la convention `convEnt` $\in$ [`convEnt1`, `convEnt2`].

2.3 Génération aléatoire d'ensembles de clauses

En s'inspirant de ce qui a déjà été fait pour les relations, écrire un programme `ensClauses = genAleatEnsClauses(nbClauses, maxNumAtome, maxLgClause)` qui produit dans `ensClauses` un ensemble de `nbClauses` clauses utilisant les atomes de p_1 à $p_{\texttt{maxNumAtome}}$ et chacune de longueur maximale `maxLgClause`.

2.4 Écriture

Écrire un programme `ecrireEnsClauses(ensClauses convSor)` qui imprime l'ensemble de clauses `ensClauses` selon la convention `convSor`, oú `convSor` est un entier dans $[0, 1, 2]$ qui designe une des 3 répresentations décrites dans 1.3.

3 Implémenter l'algorithme de Davis et Putnam

3.1 Cahier des charges

Implémenter l'algorithme de Davis et Putnam (vu en cours, remis ci-après dans ce document) pour déterminer si un ensemble de clauses est satisfaisable ou non.

On écrira un programme `algoDP (ensClauses nivTrace)` qui prendra en entrée un ensemble de clauses mis sous la forme interne précédente, et produira `oui` (satisfaisable) ou `non` (contradictoire).

On demande de prévoir un niveau de trace : 0 si pas de traces, 1 si un peu (= suivi), 2 si beaucoup (= verbeux).

Implémenter d'abord la forme récursive vue en cours (et rappelée en fin de ce document), puis si possible une forme itérative.

Bien sûr, il faudra spécifier et implémenter plusieurs procédures, une pour chaque action élémentaire comme "faire la résolution de 2 clauses C1, C2 sur un littéral h (et son conjugué)".

3.2 Expérimentation

Faire tourner cet algorithme sur quelques exemples.

On pourra en fabriquer, et on demande aussi d'en prendre dans le cours et les TD, en faisant à la main la transformation vers un ensemble de clauses contradictoire ssi l'argumentation de départ (Hypothèses $\models$ Conclusion) est valide.

3.3 Évaluation

"Instrumenter" les programmes pour pouvoir produire des statistiques sur le déroulement de l'algorithme : nombre et proportion des différentes actions, coût en temps.

4 Algorithme de Davis et Putnam (tiré du cours)

sat(F) (fonction récursive à valeurs dans oui, non):
F est un ensemble de clauses non tautologiques.
La valeur retournée par sat est non si F est contradictoire (i.e. $F \models \square$) et oui sinon (= si F et satisfaisable).
U est l'ensemble des littéraux h tels que F contient une clause réduite à h.
$Atomes$ et $Lit1$ ont déjà été définis. U^- est l'ensemble des conjugués de U.
1. Actions initiales
- si $F = \emptyset$ retourner(oui),
- sinon, initialiser $Lit1$, $Atomes$ et U à $\emptyset$.
2. Parcourir séquentiellement les clauses C de F.
- si $C = \square$: retourner(non).
- si C n'a qu'un littéral: $U := U \cup \{C\}$; $F := F - \{C\}$.
- en même temps, construire progressivement $Atomes$ (atomes de F) et $Lit1$ (littéraux unitaires de F).
3. Si $U \neq \emptyset$ (résolution unitaire)
- si U contient une paire opposée: retourner(non).
- reparcourir les clauses C de F, et
si $C \cap U \neq \emptyset$ alors $F := F - \{C\}$ sinon si $C \cap U^- \neq \emptyset$ alors $C := C - U^-$.
- $Lit1 := Lit1 - U$; $Atomes := Atomes - \{$atomes de $U\}$;
4. Si $Lit1 \neq \emptyset$ (éliminer les clauses à littéraux monoforme)
reparcourir les clauses de F:
- supprimer celles ayant un littéral dans $Lit1$.
- supprimer de $Atomes$ les atomes de $Lit1$.
5. Éliminer toute clause C de F contenant une autre clause D de F.
(éliminer les inclusions - phase omettable)
6. Si F a été modifié (dans les phases 2 à 5):
sat:= sat(F); (appel récursif).
7. Sinon, résolution sur une paire. Prendre un atome a dans $Atomes$.
Initialiser F^+ et F^- à $\emptyset$.
- Parcourir les clauses C de F, et, si C contient a [resp. $\bar{a}$]:
$F := F - C$; $F^+ := F^+ \cup (C - a)$; [resp. $F^- := F^- \cup (C - \bar{a})$.
(enlever C de F et ajouter $C - a$ [resp. $C - \bar{a}$] à F^+ [resp. F^-]).
- Calculer $res_a(F) := \{C \vee D \mid C \in F^+, D \in F^-\}$; puis $F := F \cup res_a(F)$.
8. sat:= sat(F); (appel récursif)