Projet 2 L3IF — Premier rendu

à faire en binôme,
à partir du moment où votre binôme figure dans la liste disponible sur la page du cours
à rendre pour le **18/02/2016 à 23h59**

envoyez une archive *qui compile* à `olga.kupriianova@ens-lyon.fr`,
`antoine.plet@ens-lyon.fr` et `daniel.hirschkoff@ens-lyon.fr`

Il vous est demandé, pour le premier rendu, de rendre un solveur SAT implémentant l'algorithme DPLL vu en cours.

Il vous faudra aussi, suivant votre niveau d'avancement, fournir une autre version du solveur, qui accepte des formules dans la syntaxe étendue (cf. TP numéro 1), et qui s'appuiera sur la transformée de Tseitin.

1 Solveur brut

1.1 Formats pour votre solveur DPLL

En entrée. Votre programme devra respecter **scrupuleusement** le format suivant en entrée, pour la description d'un problème SAT. Le format est organisé par lignes.

- Ligne d'*en-tête*: `p cnf V C`, où `V` est l'indice maximum utilisé pour les variables et `C` le nombre de clauses.
- Lignes représentant une *clause*: une suite d'entiers $\neq 0$ terminée par un 0 (x_3 est représenté par 3, $\overline{x_3}$ par -3).

  Exemple: `1 -9 -2 7 0` *représente* $x_1 \vee \overline{x_9} \vee \overline{x_2} \vee x_7$
- Lignes "`c xxx`" : commentaire.

Une ligne de commentaire peut apparaître n'importe où, y compris avant la ligne d'en-tête. Elle peut être vide. Les clauses apparaissent toutes après l'en-tête.

Si le `C` ou le `V` qui est annoncé dans l'en-tête du fichier ne correspond pas à ce que contient effectivement le fichier, affichez un message d'avertissement du style *"Le fichier comporte n clauses, alors que C clauses étaient annoncées."*, mais continuez quand même (à l'image de ce que fait `minisat`).

Appeler le programme. L'exécutable correspondant à votre programme devra être nommé `resol`, et il faudra qu'il puisse être appelé de la manière suivante pour lancer le programme sur le fichier `ex2.cnf`:
`./resol ex2.cnf` (pas de `resol < ex2.cnf`, ou de `./resol -f ex2.cnf`, ou autres).

Un fichier d'entrée du solveur aura pour suffixe `.cnf`

En sortie.

- Ligne "`s SATISFIABLE`" : problème satisfiable;
- Ligne "`s UNSATISFIABLE`" : le contraire;
- Ligne "`s ???`" : solvabilité inconnue;
- Votre programme devra renvoyer une affectation des variables dans le cas où la réponse est `SATISFIABLE`.

  Pour cela, après la ligne où est affiché `SATISFIABLE`, afficher la suite des valeurs associées aux variables, terminées par un zéro (exemple: `-1 2 3 4 -5 0`)[1].

[1] Vous pourrez constater que `minisat` considère que si 5 est l'index maximal des variables, toutes les variables de 1 à 5 existent, même s'il y en a qui ne sont pas mentionnées dans les clauses.

Comme indiqué maintes fois lors du premier cours, respectez scrupuleusement les indications données pour les entrées et les sorties. Cela nous permettra en particulier de soumettre vos programmes à des batteries de tests.

1.2 Algorithme DPLL

Tout le monde devra coder une version "de base" de l'algorithme DPLL vu en cours.

On se contentera d'une approche naïve pour la sélection de la prochaine variable sur laquelle miser (optez pour le plus simple, du style "la prochaine qui est inconnue", en commençant par la supposer vraie).

Si vous n'êtes pas très expérimentés en programmation, commencez par avoir un programme *qui marche* avant de songer à des améliorations.

Vous pouvez commencer par faire tourner le programme sans faire de déduction, autrement dit en faisant une bête énumération de toutes les instanciations possibles jusqu'à en trouver une qui marche.

Pour la déduction, implémentez dans un premier temps les clauses unitaires (tous les litéraux d'une clause sauf un sont à faux). Vous ajouterez ensuite la polarité unique (une variable n'apparaît qu'avec une seule polarité dans les clauses actives) : il vaut mieux rendre un programme qui marche mais ne traite pas la polarité unique qu'un programme qui essaaye de faire les deux sans y arriver.

Ces suggestions peuvent également être utiles pour concevoir le découpage du code au sein du binôme.

1.3 Tests.

Testez votre programme sur *au moins* 10 fichiers, que vous fournirez dans un répertoire dans votre archive.

Vous pouvez écrire vos premiers tests à la main, avec l'idée de vérifier que divers cas particuliers dans le comportement du solveur sont bien traités.

Vous pouvez également écrire un programme pour engendrer des tests. Dans ce cas, expliquez comment utiliser ce programme dans le fichier README (cf. ci-dessous), et n'incluez pas les tests engendrés dans l'archive que vous nous envoyez (afin de ne pas la "gonfler" artificiellement) : il faudrait que nous puissions les engendrer nous-mêmes.

. . . si les choses se passent mal, vous pouvez rendre un programme pour lequel vous pourrez indiquer quelque chose comme *"les tests 1 à 15 marchent bien, il y a un problème sur les tests 16 et 17, lié probablement à . . . "*.

2 Solveur convivial : transformée de Tseitin

Les binômes "avancés" devront également coder la transformation de Tseitin. Pour savoir si vous êtes dans un binôme "avancé", reportez-vous à la liste des binômes disponibles à partir de la page du cours (à partir du 5 février).

L'énoncé du TP numéro 1 décrit le format en entrée du solveur convivial, qui accepte une formule quelconque (non nécessairement en forme normale conjonctive).

Le suffixe `.for` sera utilisé pour les fichiers contenant des formules quelconques.

L'exécutable `resol` devra être appelé de la manière suivante :

```
./resol -tseitin toto.for
```

Les mêmes remarques que ci-dessus s'appliquent s'agissant des tests.

3 Que prendre en compte

Modularité. Découpez votre code en plusieurs fichiers (la compilation se faisant à l'aide d'un Makefile).

Outre un fichier principal contenant le cœur de l'algorithme DPLL, il faudra aussi gérer de manière "propre" l'accès aux diverses structures de données (litéraux, clauses).

Autant que faire se peut, il faudra que vous conceviez la structure générale de votre code en ayant en tête les extensions mentionnées sur les transparents du cours (voir le transparent 19).

Efficacité. Tâchez de faire un compromis raisonnable entre vos capacités en programmation et l'efficacité dans la manipulation des données. Expliquez vos choix d'implémentation dans le fichier README, et indiquez les endroits où vous êtes conscients du fait que l'efficacité pourrait être améliorée.

Pré-traitement de l'entrée. On souhaite détecter les clauses tautologiques (contenant $x_i \vee \overline{x_i}$), et les clauses unitaires. Les fonctions de propagation peuvent par ailleurs être utilisées pour le prétraitement de l'entrée. Vous pouvez également songer à des traitements plus poussés : détecter les variables apparaissant avec une seule polarité, collecter des informations sur les clauses et les variables, etc.

Traçabilité élémentaire. Munissez votre programme de quelques fonctions d'affichage pas nécessairement très sophistiquées pour pouvoir examiner ce qui se passe en cours d'exécution. Le but est de pouvoir afficher l'état du système alors que l'algorithme tourne, et ce n'est pas très grave si ce n'est pas ultra lisible. Cela pourra vous servir en particulier à développer et débugger votre programme. Expliquez dans votre rendu comment utiliser la fonctionnalité d'affichage.

4 Une "check list" pour votre rendu

- Une archive envoyée **à l'heure** aux trois encadrants, avec un nom de fichier significatif.
- Ça respecte les consignes du rendu.
- Le code est structuré de manière lisible, et commenté.
- Ça compile et fonctionne sur les machines des salles libre service de l'ENS.

  Ce point a pour conséquence qu'il faut éviter que la compilation s'appuie sur des outils sophistiqués dont vous seuls disposez sur votre machine.
- Ce fut testé (il y a un sous-répertoire contenant des fichiers de test que vous avez utilisés).
- Un fichier README contenant au moins les points suivants:
  1. Une description de la structuration du code, et des choix d'implémentation importants (en particulier, structures de données, avec éventuellement des remarques sur des améliorations que vous envisagez d'apporter).
  2. Quelques mots sur la répartition du travail pour ce rendu (qui a fait quoi).
  3. Des commentaires sur les performances que vous avez pu observer en testant votre solveur.