Projet 2 L3IF — Deuxième rendu

à faire en binôme, à rendre pour le 1er avril 2014 à 23h59
envoyez une archive qui compile à `valentin.blot@ens-lyon.fr` et `daniel.hirschkoff@ens-lyon.fr`

1 Généralités

Le but de ce rendu est d'enrichir le solveur qui a fait l'objet du rendu 1 de diverses manières. A priori, il n'y a pas de grande difficulté du point de vue du codage.

Il faudra qu'il soit possible, sans trop de difficulté, d'utiliser diverses versions du solveur, en fonction des optimisations/variantes que vous aurez codées (voir ci-dessous). Vous pourrez pour ce faire soit gérer des options que l'on peut entrer en ligne de commande, soit engendrer (un tant soit peu élégamment, *en particulier en évitant de dupliquer maladroitement de grandes parties du code*) des exécutables différents.

2 Coder: SAT, formules logiques, coloration

2.1 SAT encore: heuristiques

Le solveur devra se décliner en plusieurs versions:

- La version du rendu 1, éventuellement nettoyée de bugs que vous auriez découverts après le premier rendu.

  Pour les binômes "avancés", le rendu 1 fournit deux versions, avec et sans *watched literals*.

- Diverses versions en plus.
  - **Pour les avancés.** Les trois heuristiques vues en cours (RAND, MOMS, DLIS).
  - **Pour les non avancés.** Au moins RAND et une "optimisation" du solveur, à choisir entre les Watched Literals et une heuristique autre que RAND (MOMS ou DLIS).

  À noter au passage qu'il est demandé que RAND soit dynamique*: il ne s'agit pas de permuter aléatoirement l'entrée avant de faire tourner le programme qui parie sur la prochaine variable non assignée, mais bien de parier au hasard "en cours de route".*

- Si de votre propre chef vous implémentez d'autres améliorations, elles sont les bienvenues.

Bien entendu, toutes les versions fournies du solveur devront être testées et renvoyer des résultats corrects.

Usage. Le programme devra pouvoir être utilisé comme pour le rendu 1: même format en entrée, même format en sortie.

Chacune des versions du solveur devra s'appeler en tapant

```
machin fichier.cnf
```

où `machin` est soit un exécutable (avec un nom comme p.ex. `resol-wl`), soit un appel de la forme `./resol -option1 -option2..`

2.2 Formules logiques: transformation de Tseitin

Dans cette partie, il vous est demandé d'utiliser votre solveur SAT pour prendre en compte une formule logique quelconque.

Cela signifie qu'il vous faudra implémenter des analyseurs lexical et syntaxique afin de saisir des formules logiques quelconques, pour ensuite transformer celles-ci en formules en forme normale conjonctive.

La grammaire des formules en entrée est la suivante:

$$\mathcal{F} \quad ::= \quad \mathcal{F}_1 \vee \mathcal{F}_2 \mid \mathcal{F}_1 \wedge \mathcal{F}_2 \mid \neg\mathcal{F} \mid \mathcal{F}_1 \Rightarrow \mathcal{F}_2 \mid \texttt{x}$$

`x` désigne une *variable propositionnelle*. Concrètement, les identificateurs seront des suites de lettres minuscules, la conjonction et la disjonction seront notées respectivement `/\` et `\/`, l'implication `=>` et la négation `~` (bien entendu, les parenthèses peuvent aussi être utilisées pour écrire les formules). À vous de réfléchir aux priorités respectives des opérateurs. Les variables seront données par des suites non nulles de lettres et de chiffres commençant par une lettre minuscule.

Usage. Le programme devra pouvoir être utilisé de la façon suivante: `tseitin fichier.txt`
où `fichier.txt` contiendra une formule logique au format écrit ci-dessus. La sortie devra obéir au même format que pour les problèmes SAT (SATISFIABLE/UNSATISFIABLE, et une affectation des variables dans le premier cas).

De plus, votre programme devra pouvoir afficher la formule en forme normale conjonctive obtenue à partir d'une formule logique en entrée (affichage au format d'entrée de DPLL).

2.3 SAT pour résoudre des problèmes NP-complets
binômes avancés uniquement

On s'intéresse ici au problème de la colorabilité des graphes.

Votre programme prendra en entrée un graphe décrit suivant le format DIMACS
(voir `http://mat.gsia.cmu.edu/COLOR/instances.html` pour quelques graphes).

Il vous est demandé de faire un programme prenant en entrée un entier k et la description d'un graphe, et testant la k-colorabilité en s'appuyant sur votre solveur SAT. L'invocation du programme se fera de la manière suivante: `./colorie k fichier.col`
où `k` est le nombre de couleurs et `fichier.col` le fichier décrivant le graphe à colorier.

La sortie décrira un graphe au format `dot/graphviz`. Voir ici, et le petit exemple disponible depuis la page www du cours.

On demande aussi un retour sur la complexité observée: pour quelles valeurs de k et tailles de graphe le solveur rend-il les armes?

3 Performances

3.1 Fabriquer des moulinettes pour les tests

Une partie importante de ce rendu porte sur l'évaluation de l'efficacité des diverses déclinaisons de votre programme (pas nécessairement l'efficacité en elle-même). On se focalise ici sur la complexité en temps, vous pouvez également, si vous le souhaitez, traiter la complexité en espace.

Il vous est demandé de rendre, outre les programmes décrits à la Section 2, de petits programmes permettant de faire tourner des tests, ainsi que les résultats de ces tests, sous forme de courbes.

Pour fabriquer ces "moulinettes" de tests, vous pouvez:

- soit prendre comme point de départ l'exemple minimal qui est proposé à partir de la page www du cours (script `bash` / génération de fichier `gnuplot`).
- soit coder tout ou partie de cette composante de tests dans le langage de programmation que vous utilisez pour votre solveur SAT (autrement dit étendre le programme en ajoutant des fonctions qui appellent le solveur et stockent les observations dans des fichiers — ne vous amusez pas à reprogrammer `gnuplot` pour le tracé des courbes!).

Vous trouverez en ligne des pages de tutoriaux pour `bash` (par exemple ceci) et `gnuplot` (par exemple cela).

3.2 Des expériences

On vous demande de faire tourner des tests pour comparer l'efficacité des diverses déclinaisons de votre programme:

- le solveur SAT, dans ses différentes versions: par exemple, avec ou sans l'heuristique machin, et avec ou sans watched literals, cela donne 4 versions;
- le solveur de formules logiques;
- le colorieur de graphes *(pour les binômes avancés)*.

À vous de choisir des jeux de tests que vous jugez significatifs, d'écrire les moulinettes idoines, et de documenter les comparaisons que vous avez pu faire. La réflexion sur la construction des jeux de tests est importante — typiquement, vous devrez vous poser des questions comme *"quel genre de clauses est susceptible d'avantager/de faire souffrir telle heuristique?"*. Dans le même ordre d'idées, la présentation des résultats suivra cette logique: on évitera de mettre "numéro du test" en abscisses dans une courbe, par exemple.
Une remarque importante: les capacités des cerveaux des encadrants du cours sont telles qu'on ne sait pas regarder plus de 12 courbes. Le travail d'analyse de l'efficacité est aussi un travail de synthèse, et nous ne souhaitons pas recevoir les résultats de 100 expériences bruts de décoffrage.

3.3 Conclusions pleines de sagesse

Dans un fichier à part (appelé par exemple EXPERIENCES), vous collecterez les observations que vous pouvez faire sur les expériences que vous avez menées. Vous expliquerez vos choix, les difficultés que vous avez rencontrées, et des hypothèses permettant d'expliquer d'éventuelles sources d'inefficacité.

Bien entendu, en fonction de l'efficacité de votre solveur, vous aurez plus ou moins de latitude pour explorer des jeux de tests étendus et variés. Il vous est néanmoins demandé de calibrer vos expériences de façon à pouvoir "observer" quelque chose de pertinent (il n'est pas envisageable que votre programme ne marche uniquement que sur des problèmes à deux clauses de deux litéraux chacune).

4 Une "check list" pour votre rendu

- Une archive envoyée **à l'heure** aux deux encadrants, avec un nom de fichier significatif.
- C'est bien la dernière version du programme que vous nous envoyez.
- Ça respecte les consignes du rendu.
- Ça compile (sur votre ordinateur *et* sur les machines des salles libre-service).
- Ce fut testé.
- Le code est structuré de manière lisible, et commenté.
- Un fichier README contient les explications
  - sur d'éventuelles améliorations que vous avez apportées à DPLL depuis le rendu1 (par exemple, correction de bugs);
  - sur l'utilisation de vos programmes (comment les exécuter, comment récupérer leurs sorties);
  - sur les choix d'implémentation importants, et sur l'organisation du code;
  - sur la manière dont vous vous êtes répartis le travail (qui a fait quoi);
  - sur d'autres remarques que vous pouvez faire, par exemple s'agissant de la performance.

- Un fichier EXPERIENCES rassemblera les informations portant sur la partie expérimentale du rendu:
  - comment sont faites les moulinettes, comment on s'en sert;
  - comment ont été conçus les divers tests;
  - ce qu'on peut tirer comme conclusions.
- Les diverses courbes rendant compte de vos expériences.

5 Échéancier indicatif

Comme toujours, il vous est fortement déconseillé de commencer le travail à l'approche de la date limite. Voici une suggestion d'échéances intermédiaires:

- formules logiques = une semaine
- heuristique(s)/coloriage de graphes = une semaine
- scripts, mise en place des expériences = une semaine