Projet 2 L3IF — Deuxième rendu

à rendre pour le **dimanche 22 mars à 23h59**[1]
envoyez une archive qui compile à
`julien.bensmail@ens-lyon.fr`, `vincent.lanore@ens-lyon.fr` et `daniel.hirschkoff@ens-lyon.fr`

Ce qui suit présente les divers ingrédients du deuxième rendu.
Vous recevrez des mails binôme par binôme, vous précisant ce qu'il est attendu de vous.

1 Coloriage de graphes

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

Le fichier en entrée spécifie 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:

```
./resol -color  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. Voir aussi la page du cours pour un exemple montrant comment avoir plusieurs analyseurs syntaxiques, et comment récupérer les options, tout cela en Caml — n'hésitez pas à nous faire signe si vous souhaitez disposer de la même chose en C++.

Pour le rendu. Créez un sous-répertoire dans le répertoire de fichiers tests dans lequel vous mettrez les fichiers que vous avez utilisés pour tester le coloriage. Dans le README, décrivez brièvement comment vous avez fait l'extension de DPLL avec le coloriage, et indiquez quelques exemples à faire tourner pour voir votre programme répondre rapidement, et aussi d'autres pour le voir souffrir.

2 Apprentissage de clauses

Le traitement de l'apprentissage de clauses est décomposé en deux étapes. Nous vous recommandons d'organiser votre travail suivant cette décomposition: commencez par vous assurer que la première phase est correcte avant de passer à la seconde. Comme toujours, il vaut mieux rendre un programme qui fait bien la première phrase et ne traite pas la seconde plutôt qu'un programme qui fait tout mal.

2.1 Première phase : explorer un conflit

On lance le programme: pour cela, il faut taper `./resol -cl-interac fichier.cnf`

Le programme s'arrête au premier conflit, et propose un mode interactif. L'utilisateur peut alors taper

- "`g`" pour engendrer un fichier au format `dot/graphviz` (cf. partie sur le coloriage) décrivant le graphe des conflits. Le graphe devra mettre en évidence les nœuds qui sont dans le niveau de décision courant (en bleu), ainsi que ceux qui font partie de l'ensemble "UIP" choisi (en violet, sauf le nœud UIP qui sera en jaune), afin que la clause nouvellement engendrée soit facilement lisible. Ne dessinez pas tout le graphe des conflits, arrêtez-vous lorsque tous les nœuds bleus/violets/jaune (et leurs causes) apparaissent.
- "`c`" pour continuer jusqu'au prochain conflit (et la saisie au clavier recommence);

[1] Manifestez-vous avant le 9 mars si vous souhaitez que cette date soit repoussée d'une semaine.

- "`t`" pour aller jusqu'au bout de l'exécution sans s'arrêter.

NB : tant que vous ne traitez pas la seconde phase décrite ci-dessous, les options "`c`" et "`t`" se contentent d'ajouter la clause apprise, et on fait du backtrack chronologique (en revenant sur le dernier pari). Il vous faudra donc vérifier que l'ajout des clauses apprises ne change pas la sortie du programme (!), et aussi, si possible, regarder si l'apprentissage de clauses sans backtrack malin peut améliorer les performances ou pas.

Pouvoir débrancher le graphe. Veillez à ce que les calculs nécessaires au dessin du graphe ne soient pas faits systématiquement. Ceci afin de ne pas pénaliser les performances du solveur avec l'option `-cl` (voir seconde phase ci-dessous).

2.2 Seconde phase : apprentissage de clauses

À chaque conflit, calculez la clause que l'on ajoute, ainsi que le niveau auquel il convient de faire un backtrack.

Gardez à l'esprit que le backtrack peut possiblement se faire plus "haut" que le dernier pari ayant été effectué: si le dernier pari est au niveau 12, et si tous les littéraux de la clause que l'on rajoute sauf un sont de niveau au plus 8, on saute au niveau 8 (intuitivement, la valeur de l'unique litéral de niveau 12 aurait pu être déduite dès le niveau 8).

Dans le mode interactif, le programme affiche la clause qui est ajoutée (sous la forme habituelle, avec un zéro à la fin), et, à la ligne suivante, le niveau où il va backtracker (affichez simplement un entier).

Proposez aussi la possibilité de lancer `./resol -cl fichier.cnf`
pour lancer le solveur avec apprentissage de clauses, mais sans mode interactif.

2.3 CL et WL

Commencez par développer CL (*clause learning*) sans WL (*watched literals*), puis, lorsque cela marche, passez à CL avec WL (si cela vous est demandé, bien entendu).

2.4 Bonus

2.4.1 Viser le premier UIP

Pendant que l'on construit la clause que l'on apprend, lorsqu'il y a k litéraux qui ont été *déduits* au niveau courant avec $k \geq 2$, faire la résolution avec celui qui a été affecté en dernier. Voire faire la résolution avec les $k-1$ les plus récemment affectés, simultanément.

Ceci pour éviter de "rater" le premier UIP, en remontant le graphe des conflits de manière maladroite.

2.4.2 Prouver la non satisfiabilité

Lorsque le problème est satisfiable, le solveur peut afficher une affectation des variables qui satisfait la formule donnée en entrée.

Dans le cas contraire, on peut dériver une contradiction en utilisant la résolution.

En appelant le programme de la manière suivante

```
./resol -explainunsat
```

le solveur fournira, lorsque le problème est insatisfiable :

- un sous-ensemble des clauses du problème initial suffisant à dériver une contradiction ;
- une preuve par résolution de la contradiction. Celle-ci pourra être affichée suivant un format que vous définirez, ou alors sous forme d'arbre de dérivation en LaTeX.

Une manière simple de tester cette fonctionnalité de votre solveur, dans un premier temps, pourra être d'engendrer, à partir d'un problème insatisfiable, plusieurs copies du même problème, en décalant les indices de variables.

Pour le rendu. Expliquez clairement comment vous avez adapté le programme du rendu 1 (modification des structures de données, code additionnel) afin d'avoir ce qu'il faut pour faire l'apprentissage de clauses et dessiner le graphe de conflits.

Exemple(s) pédagogique(s). Dans votre rendu, proposez au moins un exemple servant de support à un "guide d'utilisation" de votre système d'exploration des conflits: on lance le programme, on s'arrête sur tel ou tel backtrack, et on constate que la clause ajoutée est "blabla" (tout cela étant expliqué dans le fichier README).

3 Rendu: gros fichiers

Avec votre solveur qui s'etoffe, le mail que vous enverrez pour ce rendu risque d'etre bien gros, à cause de fichiers de tests conséquents.

Merci d'anticiper la chose en évitant d'inclure les gros fichiers dans le mail. Pour cela, deux solutions, a priori :

1. Vos tests sont engendrés à l'aide d'une moulinette: ne les incluez pas dans votre archive.
2. Vous mettez une archive avec les tests sur une page www, et on utilise la commande `wget` pour recuperer l'archive (par exemple: `wget http://perso.ens-lyon.fr/daniel.hirschkoff/P2/docs/rendu2.pdf`).

Dans les deux cas, modifiez votre Makefile de facon à gerer la chose, et mettez 2 mots dans le README à ce sujet. Pour donner un exemple, ajoutez les deux lignes suivantes (attention! la seconde ligne commence par une tabulation, juste avant `echo`) au fichier Makefile :

```
exemples:
	echo "je recupere les exemples"; wget http://perso.ens-lyon.fr/daniel.hirschkoff/P2/docs/rendu2.pdf
```

En tapant `make exemples` , on pourra alors recuperer le fichier en question.

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.
- Ça compile et fonctionne sur les machines des salles libre service de l'ENS.
- Ce fut testé (il y a un sous-répertoire contenant des fichiers de test que vous avez utilisés).
- Le code est structuré de manière lisible, et commenté.
- 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).
     En particulier, si vous avez implémenté l'apprentissage de clauses, expliquez comment vous avez dû modifier votre code pour prendre en compte
     - le fait que l'on peut ajouter des clauses en cours de route;
     - la nécessité de garder une trace de la raison pour laquelle la valeur d'un litéral a été déduite;
  2. Quelques mots sur la répartition du travail pour ce rendu (qui a fait quoi).
  3. Ce que vous avez repris du rendu précédent, soit parce que vous avez trouvé des bugs, soit parce que vous avez apporté des améliorations ici ou là.
  4. Des commentaires sur les performances que vous avez pu observer en testant votre solveur.