Projet 2 L3IF — Quatrième et dernier rendu

à faire en binôme, à rendre pour le 17 mai 2015 à 23h59
envoyez une archive qui compile à
`julien.bensmail@ens-lyon.fr`, `daniel.hirschkoff@ens-lyon.fr` et `vincent.lanore@ens-lyon.fr`

1 Déclinaisons de votre solveur : heuristiques pour les paris

Suivant votre passé dans les rendus 1, 2 et 3, votre solveur peut se présenter en diverses versions : sans option, avec une combinaison des options `-wl`, `-cl` (et éventuellement d'autres options, suivant les cas).

On ajoute ici d'autres options possibles, en déclinant plusieurs manières de faire le pari dans les cas où on ne peut plus faire de déductions mais il reste des litéraux inconnus.

1. RAND

   On tire au hasard la prochaine variable sur laquelle on parie, et le pari que l'on fait (variante: compter combien de fois x et $\overline{x}$ apparaissent dans des clauses *vivantes*, et parier sur le plus fréquent).

   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".*

   option : `-rand`

2. MOMS (*Maximum Occurrences in clauses of Minimum Size*)

   On parie sur le littéral qui apparaît le plus dans les clauses de taille minimale (intuition: c'est l'endroit "le plus contraint", où la propagation va agir davantage)

   option : `-moms`

3. DLIS (*Dynamic Largest Individual Sum*)

   On parie sur le littéral le plus efficace, i.e., qui rend le plus de clauses satisfaites

   (variante: $\text{score}(\alpha) = \sum_{\alpha \in C, C \in \phi} 2^{-|C|}$, choisir α qui maximise le score).

   option : `-dlis`

4. VSIDS (*Variable State Independent Decaying Sum*) — (rappel du rendu 3)

   On associe à chaque littéral un *score*, qui augmente à chaque fois qu'il apparaît dans une clause apprise (variante : à chaque fois qu'il est manipulé dans l'analyse de conflit). On divise régulièrement par une constante (par exemple 2), pour privilégier l'activité récente.

   Lorsque l'on doit faire un pari, on choisit le littéral non affecté dont l'activité est la plus grande.

   option : `-vsids`

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

2 L'heure des bilans : performances

2.1 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 pure 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 de petits programmes (des "moulinettes") 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`).

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

- 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! — cf. ci-dessus pour un tutoriel).

2.2 Des expériences

Utilisez vos moulinettes afin de faire tourner des tests pour comparer l'efficacité des diverses déclinaisons de votre solveur SAT. À vous de choisir des jeux de tests que vous jugez significatifs, et de rendre compte des comparaisons que vous avez pu faire. La réflexion sur la construction des jeux de tests est importante — il s'agit de "mettre au point ses expériences" de manière à ce qu'elles permettent d'observer quelque chose de significatif (par exemple, valider des hypothèses sur la forme de problème qui avantage plus ou moins telle ou telle heuristique).

Vous pouvez bien sûr inclure des tests avec `minisat` quand vous faites vos comparaisons (.. sauf si cela rend vos courbes illisibles).

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).

2.3 Conclusions pleines de sagesse

Votre rendu contiendra un sous-répertoire `Exps`, contenant:

- Des fichiers `pdf` présentant des tableaux de courbes fabriquées à l'aide de vos tests (ou alors un unique fichier présentant les tableaux les uns à la suite des autres).

  Vous pouvez présenter au maximum 8 tableaux, à vous de choisir ceux qui vous semblent les plus significatifs (et vous pouvez en mettre moins si vous n'avez pas réussi à faire suffisamment d'expériences).

  Veillez à la lisibilité des courbes. En particuler, évitez de mettre "numéro du test" en abscisses dans une courbe, par exemple.

- Un fichier EXPERIENCES dans lequel vous expliquerez

  - comment les courbes ont été engendrées, et comment on pourrait les réengendrer si on voulait le faire (utilisation des moulinettes);
  - pour chacune de vos "expériences", ce que vous observez: autrement dit, à la fois le commentaire des tableaux de courbes que vous rendez, et un commentaire sur les expériences qui n'ont pas marché, ou sur celles que vous avez décidé de ne pas inclure dans les tableaux de courbes.

Deux recommandations.

- N'oubliez pas les recommandations du rendu 3 s'agissant de gros fichiers de tests.

- Pour peu que vous poussiez un peu votre solveur dans ses retranchements, les expériences peuvent nécessiter du temps pour les faire tourner. Commencez donc *tôt* à les faire, et attachez-vous à faire des scripts pour les faire tourner, de manière à pouvoir les reproduire.

3 Ajouter le simplexe à SMT

Étendez votre solveur avec le simplexe, afin de traiter l'arithmétique linéaire. Vous vous fonderez pour cela sur l'article `http://yices.csl.sri.com/papers/cav06.pdf`.

Ajoutez une option `-simplex` à votre solveur, et fournissez quelques exemples d'utilisation (vous pouvez choisir comment adapter la syntaxe adoptée pour SMT dans le rendu 3).

Traitez les inégalités larges, les inégalités strictes sont en bonus.

4 Utiliser SMT pour une preuve Coq

Le but de cette partie est d'ajouter un backend à la version DPLL(T) de votre solveur de façon à engendrer une preuve Coq correspondant à la sortie du solveur (bien entendu, le lemme Coq que l'on prouve dépend de la réponse du solveur).

Il vous est conseillé de traiter la théorie des égalités, les plus ambitieux (casse-cou?) s'attaqueront à la logique des différences (vous avez le droit dans ce cas d'introduire des limitations sur la forme de l'entrée du solveur).

La sortie de votre programme sera un fichier `.v` que l'on peut soumettre à Coq (par exemple avec `coqc`). Il n'est pas demandé que la preuve produite soit particulièrement élégante ou concise (c'est en quelque sorte de l'"assembleur de preuve").

Vous devrez sans doute développer une librairie Coq d'"outils" (définitions, lemmes) nécessaires à la preuve.

Il vous est conseillé d'introduire des structures de données intermédiaires (un peu à la manière des structures intermédiaires que l'on trouve dans les compilateurs, pour ceux qui ont une idée de la structure d'un compilateur). Par exemple, l'exécution de DPLL(T) peut engendrer une "trace" qui sert à justifier la sortie (SAT/UNSAT), et ensuite on transforme cette trace en quelque chose de plus structuré, pour enfin engendrer la séquence de tactiques Coq.

En *bonus*, plus ardu d'un point de vue technologique, sans que ce soit particulièrement intéressant pédagogiquement : faire une tactique Coq, appelée `resol`, que l'on peut appeler dans Coq pour traiter un but ayant la bonne forme. Vous pouvez vous aider de `http://gallium.inria.fr/blog/your-first-coq-plugin/`. Faites signe à D. Hirschkoff si vous vous lancez dans cette direction, on pourra éventuellement exploiter un "consultant Coq" pour vous aider dans les aspects un peu scabreux d'interfaçage avec Coq.

Ajoutez une option `-coq` à votre solveur. Proposez dans votre rendu au moins 4 exemples relativement simples d'utilisation de cette option (2 SAT, 2 UNSAT).

5 Une "check list" pour votre rendu

Vous savez désormais ce qui est attendu dans un README. N'oubliez pas que vous devrez aussi rendre des explications s'agissant de la partie sur les expériences (partie 2).

Il s'agit de votre dernier rendu: c'est celui-ci qui fera l'objet de l'étude la plus attentive de la part des encadrants du cours. Il faut donc que cela s'approche autant que faire se peut d'un *produit fini*. Soignez tout particulièrement le code, les commentaires dans le code, éliminez le code mort que vous auriez tout simplement commenté, supprimez les fichiers inutiles. Reprenez les explications fournies précédemment sur la structuration du code, la répartition du code, les qualités et défauts de votre solveur, etc.

Ajoutez au README une rubrique *"Critique"*, où vous indiquerez ce qui pourrait être amélioré dans votre programme. Cela peut être une structure de données qui mériterait d'être reprise car elle est une importante source d'inefficacité, ou alors une manière différente de structurer votre logiciel, etc. Bref, ce que vous n'avez pas le temps de reprendre mais que vous feriez différemment si vous deviez commencer le projet 2 demain.