Rapport projet 2, TSAT - Solveur SAT/SMT

T. Stérin, A. Torres

Mai 2016

Résumé

Réalisé à l'ENS Lyon pour l'année scolaire 2015-2016 et dans le cadre de la matière "Projet2" TSAT est un solveur SAT/SMT à l'instar de minisat [1] qui a pour but de satisfaire des formules logiques qui lui sont données en entrée. TSAT est écrit en C++ par Tristan Stérin et Alexy Torres–Aurora-Dugo. Différentes méthode de résolution ont été implémentées dans TSAT :

— DPLL (Davis–Putnam–Logemann–Loveland) standard [2].
— DPLL avec méthode des litéraux surveillés [3].
— DPLL avec méthode avec apprentissage de clause [4].
— DPLL(T) dans le cadre SMT avec les théories de l'égalité, de la congruence [5] et de la logique des différences.

1 Organisation du code

L'organisation du projet est la suivante :

```
src/
├── BETHeuristic (Classes contenant les stratégies de pari)
├── Global (Fichiers source contenant les paramètres du programme)
├── NewCore (Noyau TSAT, structures de clause)
│   └── SMT (Noyau SMT)
├── Parser (Parseurs CNF et logique)
├── RandomSatExpGenerator (Générateur de formules SAT)
├── SMTGenerator (FGénérateur de formules SMT)
├── main.cpp (Fichier source principal du solveur)
├── main.h (En tête du fichier main.cpp)
└── Makefile (Fichier de compilation du solveur)
```

2 Résumé des features

2.1 SAT

— Gestions des formules au format DIMACS/FOR.
— DPPL standard, Watched Literals, Clause Learning avec interaction possible.
— Heuristiques : RAND, MOMS, DLIS, VSIDS.
— Génération de formules aléatoires.

2.2 SMT

— Égalité **avec clause learning SMT**.
— Congruence (QF UF) **sans clause learning SMT**.
— Différences ?
— Possibilité de désactiver le clause learning SMT pour comparer les performances (-disable_smt_cl).
— Génération de formules aléatoires.

2.3 SMT

3 Choix d'implémentation

Le patron de conception en statégies à été retenu pour le développement des heuristiques de pari. Cela permet d'ajouter une heuristique sans modifier le noyau SAT. Cette méthode de conception permettrait aussi de modifier l'heuristique de pari durant le traitement de la formule.

Nous avons aussi développé les différents parseurs de formule à l'aide de ce patron de conception afin de faciliter l'ajout de nouveaux types de parseurs.

Les solveurs SMT sont aussi construits selon cette idée.

Cela a beaucoup évolué au cours du projet. Nous avons trouvé d'ailleurs que c'était l'une des difficultés du cours : faire une architecture assez maléable pour intégrer facilement des ajouts futurs. En pratique nous avons souvent ré-écrit tout le code (3 fois). L'architecture finale se veut la plus simple possible avec une utilisation massive d'unordered set et d'unordered map.

4 Détection de l'UIP

Nous avons implémenté un algorithme linéaire trouvé pour l'occasion. Étant donné un DAG à une source et un puit on souhaite trouver les noeuds tels que tous les chemins de la source au puit passe par eux : les UIP. On cherche plus particulièrement celui qui est le plus proche du puit. On s'appuie sur la remarque suivante : tous les UIP sont sur le plus court chemin entre la source et le puit.
Nous sommes dans un cas particulier où le puit a exactement deux pères. Pour

ces deux noeuds on calcule le plus haut noeud du plus court chemin auquel il peut remonter sans passer par les arêtes du plus court chemin. Le plus haut des deux est le 1-UIP. Cet algorithme est linéaire.

5 Critique des performances

5.1 SAT

La méthode avec apprentissage de clause permet d'obtenir de très bons résultats.

Pour les heuristiques, les meilleurs résultats s'obtiennent avec l'heuristique MOMS.

Les optimisations à la compilation (-O3) permettent elles aussi de rendre le solveur beaucoup plus rapide (jusqu'à 70% plus rapide dans certain cas).

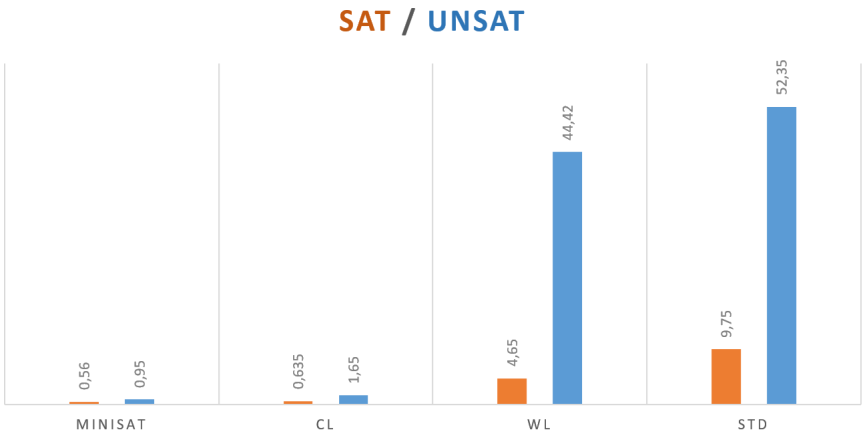

FIGURE 1 – Comparaison des temps d'execution de TSAT à MINISAT

5.2 SMT

Dans le cadre de l'égalité on propose de désactiver l'apprentissage de clause SMT. On constate par exemple sur le test gros.for l'efficacité de l'apprentissage SMT. Ce test est résolu en 15s avec apprentissage SMT et ne termine pas sans.

5.3 Remarques SMT

En cas de satisfiabilité SMT le solveur renvoie :

— Égalité : un nombre minimal de classes d'équivalences incompatibles.

— QF UF : un nombre minimal de classes d'équivalences de termes incompatibles. On prend en compte ici tous les termes qui apparaissent dans la formule (y compris les sous termes).

6 Améliorations possible

Plusieurs possibilités d'améliorations sont envisageables :

— Une parallélisation des tâches pourrait faire gagner du temps. On pourrait par exemple exécuter en même temps différentes heuristiques de pari sur la prochaine variable à assigner.

— Il faudrait implémenter le clause learning pour la congruence décrit dans [5].

— Le code pourrait être encore allégé, mais encore une fois, on s'en rend compte toujours après l'avoir écrit !

Références

[1] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[2] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[3] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[4] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[5] Albert Oliveras Robert Nieuwenhuis. Proof-producing congruence closure. echnical University of Catalonia, Jordi Girona 1, 08034 Barcelona, Spain.