Project Logique : part 2

Serge Durand

Mai 2020

1 Introduction

J'ai implémenté trois versions d'un SAT solveur : une version itérative de l'algorithme Davis-Putnam-Logemann-Loveland, une version avec le mécanisme des deux littéraux observés (*Two Watched Literals*) puis une version avec une heuristique pour le choix des variables. Pour les encodages je n'ai pu faire que le carré latin et le carré greco latin.

2 Architecture

J'ai adopté la convention suivante pour les valeurs logiques :

- 0 si l'interprétation de la variable est F
- 1 si l'interprétation de la variable T
- -1 si la variable n'a été assignée aucune valeur

Les mêmes conventions sont valables pour les littéraux. Pour les clauses :

- 0 si tout les littéraux de la clause valent 0
- 1 si au moins un littéral vaut 1
- -1 si aucun littéral ne vaut 1 et au moins un littéral vaut -1

Ainsi une clause non satisfaite mais encore potentiellement satisfaisable aura la valeur -1.

2.1 Variables

La classe *Variables* a un tableau de n entiers dans $\{-1, 0, 1\}$ en attribut, contenant la valeur des variables sous l'interprétation courante. De plus elle a un tableau de n ensembles d'entier *HashSet<Integer>*, qui sert à stocker les identifiants des clauses où la variable apparaît, pour chaque variable. J'ai fait le choix d'un tableau puisque pour une formule donnée le nombre de variables n'est pas amené à être modifié et on peut ainsi accéder en temps constant à chaque variable et à ses clauses. L'indice servant d'identifiant de la variable. A priori l'ordre des clauses dans lesquelles la variable apparaît n'a pas d'importance, mais on veut pouvoir modifier rapidement à chaque suppression/ajout de clause, d'où le choix du *HashSet*.

2.2 Littéraux

Pour les littéraux j'ai fait une classe *Literal* représentant un littéral. Elle contient l'index de la variable correspondante et un ensemble d'entiers correspondant aux clauses où le littéral apparaît comme pour les variables. De plus j'associe à chaque littéral une formule, pour avoir accès à l'interprétation courante.

2.3 Clause

Une clause dispose d'un ensemble d'entiers correspondant aux identifiants des littéraux présents dans la clause. Le choix de l'ensemble permet d'éviter les doublons dès la création de la formule. L'ordre n'est pas garanti mais n'importe pas a priori. L'accès, l'ajout et la suppression d'un littéral donné sont donc en temps constant.

La clause est également associée à une formule.

2.4 Formule

La formule contient un attribut *Variables* contenant l'interprétation courante, un tableau de $2 * n$ littéraux et un dictionnaire (*HashMap<Integer,Clause>*) contenant les clauses. Le tableau contient tout les littéraux possibles, de 0 à $n-1$ les littéraux de la forme x_i et de n à $2n-1$ les littéraux $\neg x_i$. Pour une raison qui m'échappe j'ai choisi de les stocker comme un miroir : $[x_0, x_1, x_2, \neg x_2, \neg x_1, \neg x_0]$. Cela n'apporte rien de particulier. Pour le choix du tableau c'est comme pour les variables : en principe nous ne seront pas amener à ajouter des littéraux.

Pour les clauses j'ai choisi un dictionnaire puisqu'on veut pouvoir accéder en temps constant à une clause donnée, l'ordre des clauses n'important pas, et on veut pouvoir modifier notre ensemble de clauses à faible coût (suppression et ajout répété de clauses).

De plus la formule dispose d'une classe ClausesStatus contenant deux ensembles d'entiers : l'un pour les clauses actives et l'autre pour les clauses inactives. Une clause est active si et seulement si elle est encore satisfiable, mais non satisfaite.

La présence d'un attribut formule pour les clauses et littéraux permet d'accéder à l'interprétation courante.

3 DPLL

Je me suis inspiré de [Hor19] pour la version itérative de l'algorithme Davis-Putnam-Logemann-Loveland. Essentiellement il consiste en :

1. Choisir une variable non assignée. S'il y en a aucune disponible renvoyer SAT
2. Donner la valeur 0 à la variable
3. Supprimer les clauses contenant le littéral positif suite à cette nouvelle interprétation
4. Supprimer le littéral négatif de toute les clauses qui le contiennent
5. S'il y a un conflit retourner à 2 en donnant la valeur 1 si ce n'est pas déjà fait. Si les deux valeurs ont été testées backtracker. Si ce n'est plus possible de backtracker retourner UNSAT
   (a) Tant qu'il y a des clauses unitaires reprendre comme à partir de 2 avec la variable et la valeur induite par la clause unitaire.
   (b) S'il y a un conflit annuler toutes les assignations des variables apprises par les clauses unitaires puis retourner à 2 en donnant la valeur 1 ou backtracker.
   (c) S'il n'y a aucun conflit reprendre à 1

Pour gérer le backtracking on utilise une pile avec un checkpoint pour distinguer les variables décidées des variables apprises (dont l'assignation est forcée car présente dans une clause unitaire).

On fait des désactivation / réactivation de clauses à la volée lors des phases de propagation unitaires et de backtracking.

Le choix des structures de données permet d'éviter le parcours de toute la formule aux étapes 2 et 3.

4 2-watched literal

Dans la version *2 watched literals* on associe à chaque clause 2 littéraux observés.
Pour un littéral négatif l la propagation unitaire consiste alors à :

1. Choisir une clause qui observe l
   (a) Si le deuxième littéral l_2 observé est assigné à 1 revenir à l'étape 1
   (b) Sinon parcourir les autres littéraux
      i. Si un autre littéral n'est pas assigné à 0, il devient observé à la place de l
      ii. Si aucun autre littéral n'est pas faux :
         A. si l_2 n'est assigné à 0 on est dans une clause unitaire. On fait l'assignation puis on continue à l'étape 1.
         B. Si l_2 est assigné à 0 il y a conflit : on arrête la propagation

L'algorithme final consistant alors à répéter ce processus à partir d'un littéral décidé puis de tout les littéraux appris tant qu'il y en a et de choisir (arbitrairement) un nouveau littéral non-assigné. Le mécanisme des littéraux observés maintient l'invariant suivant : un littéral observé n'est faux que si la clause qui l'observe est satisfaite.

Le backtracking consiste alors simplement à annuler les assignations, il n'y a pas besoin de changer les littéraux observés.

5 Heuristique de choix de variable

Dans les deux premiers algorithmes lorsque l'on choisit une variable non assignée le choix est arbitraire [1]. Afin d'améliorer les performances du solveur on implémente une heuristique pour le choix des variables en fonction du nombre de conflit dans lesquelles elles sont impliquées.

On maintient un ensemble de variable prioritaire et un tableau qui contient le nombre de conflit pour chaque variable, que l'on met à jour à chaque conflit rencontré. Si le nombre de conflit d'une variable dépasse un certain seuil elle est ajouté dans l'ensemble des variables prioritaires.

Puis à chaque décision on cherche d'abord à choisir une variable dans l'ensemble de variable prioritaire et s'il est vide on fait un choix arbitraire comme précédemment.

L'ensemble des variables prioritaires n'est pas trié, une variable ne devient pas plus prioritaire que les autres une fois le seuil dépassé.

Il reste alors à choisir un seuil intéressant. Sans faire une analyse très fine on a cherché expérimentalement le meilleur seuil en fonction du nombre de clauses de la formule. L'intuition étant qu'un bon seuil doit a priori dépendre de la taille de la formule. Au final on a fixé le seuil à $\frac{nombre\ de\ clauses}{17}$.

Les gains de performance ne sont pas évident sur les tests proposés. Le seul gain semble être sur *aim-50-1_6-no-1* mais il y a une perte sur *ais12*. Par contre il y a de gros gains sur les formules de carré latin, à partir de $N = 13$.

S'il y a peu de variables impliquées dans des conflits où s'il y en a trop l'algorithme revient quasiment à la version précédente, avec les étapes de calculs de conflits et de choix dans les variables prioritaires en plus.

En restant sur le même principe l'heuristique pourrait sans doute être améliorée par un choix plus fin du seuil. Il doit probablement être optimisable selon le type de problème encodé par la formule.

1. On utilise un HashSet pour suivre les variables décidées, le choix n'est donc pas totalement déterministe, le HashSet ne garantissant aucun ordre. Cependant il n'y a pas d'utilisation de fonction de tirage aléatoire non plus et en pratique les variables sont choisies quasiment dans le même ordre d'une exécution à l'autre. L'ordre étant simplement l'ordre dans lequel elles sont stockées dans la formule

Références

[Hor19] Martin Horenovsky. Modern sat solvers. https://codingnest.com/modern-sat-solvers-fast-neat-and-underused-part-3-of-n/, 2019. Accessed : 2020-14-05.