Rapport sur Projet 2 - SAT solveur

Clément Chouteau
Enguerrand Prebet
17 mai 2016

Objectif *Développer un logiciel avec travail modulaire et en équipes sur un sujet demandant un effort algorithmique, des choix d'implémentations et une validation de ces choix. Cette année, les étudiants devront développer un solveur SAT avec pour exemple minisat [1]. Le langage de programmation est laissé à leur libre choix.*

Table des matières

1 Généralités concernant le projet **2**
1.1 Utilisation du langage C++ 2
1.2 Outils utilisés 3
1.3 Application resol 3

2 Architecture du projet **4**
2.1 Tests de non régression 4

3 Notre implémentation de DPLL **5**
3.1 Le premier rendu, un algorithme paresseux 5
3.2 DPLL - structures de données 5
3.3 DPLL - fonctions principales 5

4 Heuristiques et autres **6**
4.1 Les heuristiques -rand -moms -dlis 6
4.2 L'apprentissage de clause pour améliorer DPLL 6
4.3 Implémentation d'un solveur modulo une théorie 7
4.4 Tentative de visualisation 2D des performances 8

1 Généralités concernant le projet

1.1 Utilisation du langage C++

Nous avons utilisé le langage C++ (avec des fonctionnalités de C++11) pour ce projet. Le code ci-dessous est un bon exemple d'utilisation des fonctionnalités du C++11.

ligne 3 on utilise std::map<X, Y>, une classe de la librairie standard implémentant un dictionnaire.

ligne 6 on itère par référence constante sur un conteneur.

ligne 12 on définit la fonction lambda `cmp`.

ligne 13 on utilise un algorithme de la bibliothèque standard max_element avec la fonction `cmp`.

Tout cela permet de rendre le code court, lisible et efficace.

```
1  int Formula::get_dlis_var() const
2  {
3    /* On cherche l'assignation qui rend le plus de clauses
        valides */
4    map<int, unsigned int> literals_score;
5
6    for (const Clause& clause : clauses_alive)
7      for (int l : clause.get_vars(assignment)) // chaque
          littéral doit apparaître de façon unique dans une clause
8        literals_score[l]++; // 'l' satisfait 'clause'
9        /* remarque : si variables_score[l] n'existe pas encore
            il est initialisé à 0 */
10
11   /* On récupère le littéral de meilleur score */
12   auto cmp =[](const pair<int, unsigned int>& p1, const pair<
        int, unsigned int>& p2) { return p1.second < p2.second; };
13   const auto& max_score = max_element(literals_score.begin(),
        litterals_score.end(), cmp);
14
15   int max_score_l = max_score->first;
16
17   return max_score_l;
18 }
```

Nous avons adopté autant de conventions que possible pour l'écriture du code, par exemple les noms de classes commencent par une majuscule (comme Clause) et les noms de variables par une minuscule, les noms sont séparés par des underscores. L'ordre des `#defines` est également codifié.

1.2 Outils utilisés

— Le projet est versionné avec l'outil git, Clément synchronise automatiquement son odrinateur fixe et son ordinateur portable avec l'application dropbox. La synchronisation entre Clément et Enguerrand est effectuée grâce au stockage du projet sur github.

— La compilation de l'application se fait automatiquement avec un fichier Makefile, une gestion intelligente des dépendances permet de ne recompiler que ce qui a changé. Le fichier Makefile possède d'autres options pour recompiler tout le projet avec certains paramètres comme lancer `regression`.

— Quand la taille du projet a augmenté, un éditeur de texte avancé (Code : :Blocks ou QtCreator) est devenu indispensable pour rechercher intelligemment les utilisations de variables, renommer, etc.

1.3 Application resol

Notre application qui est nommée `resol` s'appelle en ligne de commande avec le nom du fichier `exemple.cnf` donné en argument, des options sont disponibles en argument (versions courte ou longue) avec notamment `-h` alias `-help` qui permet d'afficher la liste des options disponibles en argument.

```
1  clem120@crunchbang:~/Dropbox/Projet-2 - Solveur SAT/tests$
       ./../bin/resol -h
2  Utilisation: resol [OPTION]... [FILE(.cnf|.for)]
3
4    -t  --tseitin         use tseitin transform to accept user-
       friendly .for file
5    -smte                 use SAT modulo equality theory
6    -d[N]  --debug        print trace of the SAT resolution
7    -h  --help            print help and exit
8    -wl                   activate watched litterals
9    -cl                   activate clause learning
10   -cl-interac           activate clause learning and enable
       conflict graph visualization
11   -rand -moms -dlis     use heuristic
12   -vsids                clause learning only heuristic
13
14 N is an integer between 0 (by default, no output) and 3
       setting the debug level
15   (notice : debug level is incremental)
16   3: all debug messages printed
17   2: without display during the main algorithm
18   1: without display of the formula before and after the
       pretreatment
19   0: no debug message printed
```

```
20 The option --debug sets debug level to maximum
21
22 Input file name must ends with .cnf or .for.
```

Dans le code les options sont gérées par une variable globale (mais isolée dans le `namespace` de nom `Global::`). L'affichage des messages / debug / erreurs est centralisé et géré simplement (voir exemple ci dessous).

```
1 Global::ERROR() << "cannot␣open␣file:␣" << file_name << endl;
2 Global::DEBUG(3) << "Input␣read,␣f␣is␣";
```

D'après le code au dessus l'option de ligne de commande `-dx`, `x` doit valoir au moins 3 pour que le message `Input read f is` s'affiche.

2 Architecture du projet

2.1 Tests de non régression

Nous avons écrit un programme effectuant des tests de « non régression » automatisés. En une commande `./regression.sh` et en quelques secondes on teste le programme avec diverses options sur des exemples minimaux. En cas d'erreur, un message indique sur quel type de test notre solveur plante. Cela permet de vérifier une apparition de bug impromptue et d'avoir une idée sur la cause du plantage. L'outil `regression` commence par tester le programme de façon exhaustive avec toutes les combinaisons d'options possibles sur des exemples minimaux puis il lance `more_regression` qui teste le programme sur beaucoup plus d'exemples, ce deuxième test est beaucoup plus long.

```
1 clem120@crunchbang:~/Dropbox/Projet-2 - Solveur SAT/tests$
     ./regression.sh
2 [  ] 1.cnf, expected: SATISFIABLE, "petit test 2 clauses, 3
     variables"
3 [  ] 1.for, expected: SATISFIABLE, "6\/7"
4 [ -wl ] 2.cnf, expected: SATISFIABLE, " formule vide"
5 [-cl -wl -moms] 4.cnf, expected: SATISFIABLE, "Backtrack"
6 [-cl -wl -dlis] 6_1.cnf, expected: SATISFIABLE, " test
     minimal échappant au prétraitement"
7 Press [Enter] key to start more regressions (may take a
     while)...
8
9 TESTING : more_regression/a-20-91
10 ...
```

3 Notre implémentation de DPLL

3.1 Le premier rendu, un algorithme paresseux

On a commencé à implémenter un algorithme de backtracking simple fonctionnant sur l'invariant suivant :

— strictement avant la clause `k` tout est satisfait.

— strictement avant la position `i` dans la clause `k` tous les littéraux sont faux.

L'algorithme se contentait d'avancer et de reculer dans la formule. Mais les performances étaient souvent très décevantes, le code a dû subir une refonte générale avant d'implémenter le vrai fonctionnement de l'algorithme DPLL.

3.2 DPLL - structures de données

formule Liste chaînée (`std::list`) de clause pour pouvoir insérer ou supprimer des clauses n'importe où en $O(1)$.

clause Sans l'option `-wl` on a fait le même choix que pour les clauses : liste de littéraux. Avec l'option `-wl` on stocke une clause comme un tableau (`std::vetor`) de littéraux avec 2 indices pour symboliser les littéraux surveillés. Dans ce cas les invariants sont : en dehors de ces indices tout les littéraux sont faux. Les littéraux surveillés permettent la détection des erreurs/clauses unitaires en temps constants. backtrack simple : vérifier la validité de l'invariant.

littéral Type entier signé (`int`) dont nous utilisons les valeurs x pour désigner une variable et $-x$ pour désigner sa négation.

renommage Afin d'éviter de gâcher de la mémoire dans le cas d'une formule contenant un ensemble de variables « creux » la classe `Renaming` gère le renommage des variables $(1000, 1 \to 1, 2)$.

- Gestion du backtrack : pour les décisions : pile contenant la raison = INFER/GUESS et l'affectation INFER/GUESS pour détecter l'arrêt, l'affectation pour savoir quoi annuler

pour les littéraux au sein d'une clause : pile simple

pour les clauses validées (supprimées) : tableau de listes pour un retour rapide (concaténation à la liste principales en $O(1)$)

3.3 DPLL - fonctions principales

Les étapes de l'algorithme DPLL sont :

1. Modifier les clauses du point de vue des AFFECTATIONS. Si un littéral est vrai, on enlève la clause, sinon on enlève le littéral. C'est l'objectif de `apply_modification`
2. Déduction à partir des AUTRES PARAMÊTRES : taille de la clause (`propagation_unitary`), polarité (`propagation_unique_polarity`) entraîne la découverte d'une erreur/de la fin
3. À la toute fin, une vérification complète de la solution trouvée (si elle existe) comme dernier test préventif pour la correction de l'algorithme (qui fût utile pour détecter les erreurs)

4 Heuristiques et autres

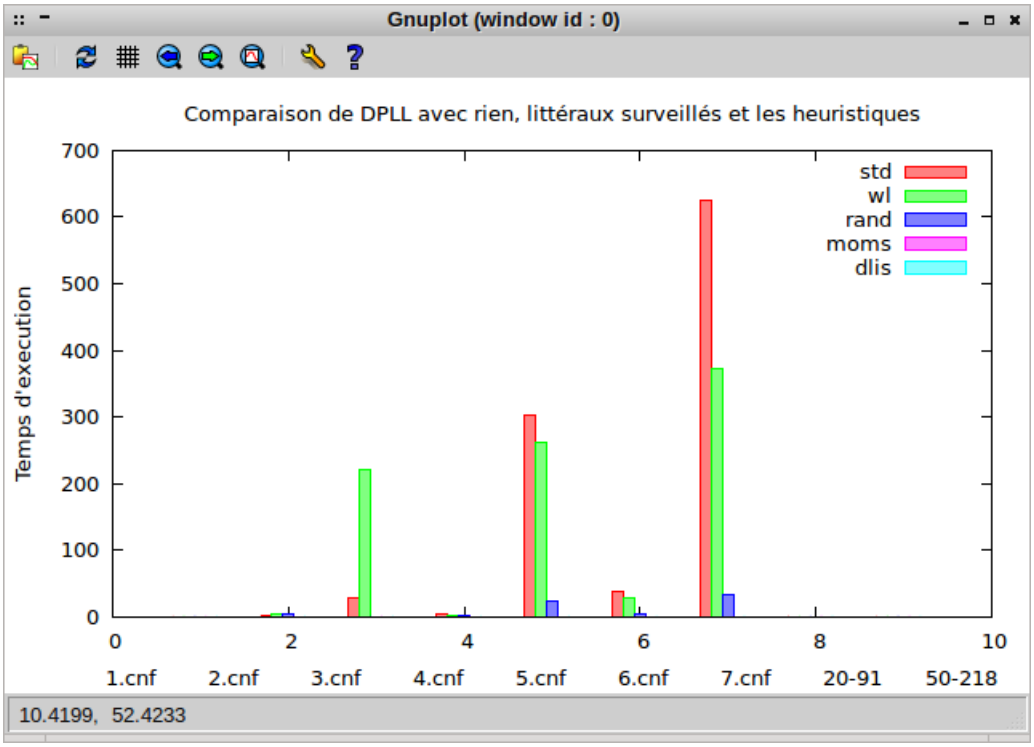
4.1 Les heuristiques -rand -moms -dlis

Une approche paresseuse est utilisée pour les heuristiques, le calcul de la variable à affecter est effectué au dernier moment, lors de l'appel de l'heuristique, un profiling précis du code nous a permis de remarquer que le temps pris par les heuristiques est négligeable dans le temps d'exécution, le choix paresseux était donc le bon.

`-rand` performances aléatoires, surprenantes, souvent rapide, rarement pire que l'algo classique.

`-moms` très bonne heuristique, en une vingtaine de lignes de codes cette heuristique permet de gagner en performances jusqu'à un facteur 100 sur certaines instances.

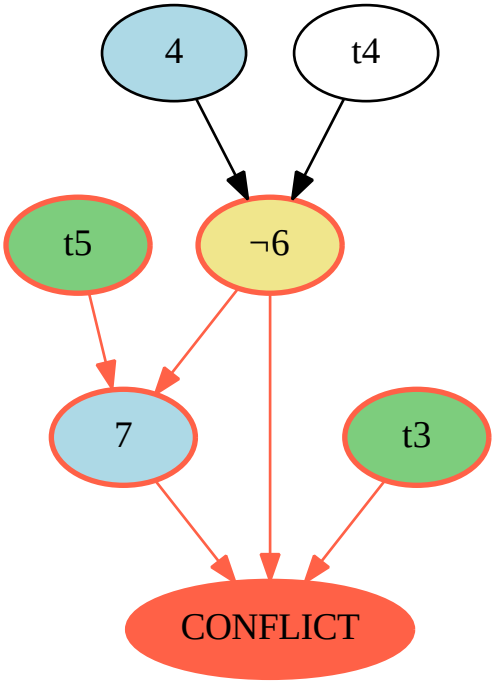
`-dlis` très semblable à `-moms`



4.2 L'apprentissage de clause pour améliorer DPLL

Lorsque l'option `-cl` est activée et qu'un conflit est détecté, l'algorithme utilise le graphe des conflits afin de générer une clause conséquence de la formule initiale qui explique le conflit. Pour cela, il faut rechercher l'UIP (Unique Implication Point), ce qui s'effectue de la manière suivante :

1. On génère le graphe du conflit en partant du conflit lui-même et en remontant au pari de l'état courant. Ainsi, les déductions faites à l'état courant mais qui n'interviennent pas dans le conflit ne sont pas prises en compte.



LÉGENDE DU GRAPHE

- Chaque noeud représente un littéral, les flèches indiquent une déduction.
- Le symbole ¬ indique qu'il s'agit de la négation de la variable.
- Le symbole `t` indique que la variable a été ajoutée par l'algorithme tseitin et n'était pas dans la formule initiale (indexation différente).

Le code couleur est le suivant :

- état courant
- état antérieur
- lié conflict
- UIP
- utilisé dans la clause apprise

2. On précalcule en $O(n^2)$ (où n est le nombre de sommets considérés), à l'aide d'une matrice `dependance` qui indique pour une case (i, j) si $i \rightsquigarrow j$, une matrice `obligation` qui indique pour (i, j) si tout chemin de i à 0 passe par j. Ainsi si le pari est sur la variable i, j est l'UIP si `obligation[i][j] == true`. On trouve alors le 1-UIP en effectuant un parcours BFS à partir du conflit.

L'option en ligne de commande `-cl-interac` permet de voir le graphe de conflict.

4.3 Implémentation d'un solveur modulo une théorie

Notre projet implémente également un solveur modulo une théorie : l'égalité.

- Dans le code un type « variant » stocke les littéraux de la théorie, donc des égalités ou des entiers (pour une formule classique) (`Real_Value`).
- L'implémentation de la théorie de l'égalité se base sur un union find pondéré par la taille (sans compression de chemin), c'est un compromis pour que chaque étape de backtrack se fasse en $O(1)$, ainsi on peut utiliser la même structure de donnée (efficace) que ce soit en mode en ligne.
- On stocke en parallèle un graphe où l'on stocke les égalités (non orienté) : $(u, v) \in E$ si la variable $(u = v)$ est vrai. De même, il y a un chemin $u \leftrightsquigarrow v$ si et seulement si $u = v$ peut être déduit par la théorie. Un conflit pour la théorie correspond toujours à une égalité fausse $(u \neq v)$ et un ensemble d'égalités qui font que $u = v$ pour la théorie. Ainsi, à partir de la différence $u \neq v$, il suffit de chercher un chemin $u \rightsquigarrow v$ dans ce graphe. La suite d'arêtes qui en découlent permet de prouver $u = v$.

Cet ensemble d'arêtes ainsi que $u \neq v$ ne peuvent pas être vrai en même temps, ce qui nous donne la clause apprise.

4.4 Tentative de visualisation 2D des performances

- On a commencé à mettre en place un générateur d'instances aléatoires de problème SAT afin de dessiner un graphique des performances du programme en fonction du (nombre de clauses / variables par clauses). pour observer les performances des heuristiques selon le nombre de clauses, de variables. Sur les graphes obtenus il est a priori impossible de distinguer les heuristiques.

1. on pense que le temps d'exécution de la plupart des tests est dominé par le traitement de l'entrée / l'initialisation (on a fait attention à ce que les IO ne sont pas pris en compte en mesurant le temps `user`).
2. il est difficile d'obtenir aléatoirement des formules « intéressantes », la plupart des formules sont trivialement satisfiables et celles qui ne le sont pas sont trop rares pour être détectées avant de tomber sur des tests trop longs pour notre algorithme.
3. il y a des chercheurs qui ont trouvé des heuristiques (sur par exemple le nombre de variables par clauses en fonction du nombre de clauses) pour trouver des instances intéressantes.

Références

[1] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.