Projet DPLL

Antoine Gontard antoine.gontard@inria.fr
Raphaël Berthon rberthon@lmf.cnrs.fr

Le projet DPLL a pour objectif de continuer sur ce qui a été fait durant le projet Rocq. Pour cela, nous allons partir d'une implémentation naïve en caml de l'algorithme DPLL et implémenter deux optimisations.

1. DPLL

1.1. Principe de l'algorithme

La façon la plus naïve de résoudre un problème SAT est de deviner la valeur des variables une à une, comme nous l'avons fait en premier lieu lors du projet Rocq. Ceci est cependant très inefficace: après avoir deviné par exemple que la variable `x` est fausse, il y a typiquement plusieurs autres variables dont la valeur est forcée par les clauses. Ceci est l'idée principale derrière l'algorithme DPLL, dont nous avons implémenté une première version à la fin du projet Rocq.

Une version basique de DPLL, après avoir deviné une variable, simplifie la formule en prenant cela en compte. Les déductions que l'ont fait sont :

- Une clause ayant un littéral vrai n'a plus besoin d'être considérée
- Tout littéral faux peut être supprimé des clauses
- Si une clause est vide, la dernière supposition est absurde, et on doit revenir en arrière
- Si une clause contient un unique littéral, on peut ajouter ce littéral à notre affectation partielle (ceci n'était pas fait dans le projet Rocq)
- Si une variable apparaît uniquement en positif ou négatif, on peut ajouter le littéral correspondant à notre affectation partielle (ceci ne sera pas fait dans ce projet).

1.2. Naïve

Une implémentation naïve de l'algorithme DPLL est donnée dans le fichier `src/naive.ml` (dans l'archive pour les encodage SAT). Lorsqu'un modèle est trouvé la fonction lève une exception pour remonter rapidement les appels récursifs, et retourne simplement lorsqu'un conflit est trouvé. L'affectation partielle des variables est représentée par une liste d'associations, de type `(literal * bool) list`.

Lorsque la variable d'environnment TRACE est affectée, l'exécution du solveur naïf produit une trace des affectations partielles successives. Vous devrez conserver cette fonctionnalité dans vos versions modifiées du solveur.

2. Améliorations

2.1. Implémentation impérative

Le solveur peut être amélioré en utilisant deux tableaux, le premier indique si une variable est utilisée ou non, et le second donne la valeur à laquelle elle est affectée le cas échéant. Cette implémentation permet de vérifier et modifier la valeur d'une variable en temps constant.

L'ennui de cette structure est qu'elle est mutable et non persistante. Ainsi, là où la fonction DPLL prenait simplement une affectation comme argument et donne des versions modifiées lors des appels récursifs, avec la nouvelle structure on doit gérer le backtracking à la main. Deux options s'offrent à nous:

- On pourrait dupliquer le tableau à l'aide d'`Array.copy`, mais ce serait très inefficace en temps comme en mémoire.
- Une meilleure approche (et donc celle que l'on va adopter) est de maintenir une unique affectation partielle (deux tableaux) qui seront utilisés par tous les appels récursifs de `dpll`, ainsi qu'une pile contenant les modifications effectuées afin de pouvoir les annuler. Quand une variable `x` est affectée, on ajoute la valeur `x` à la pile. Une valeur spéciale, par exemple `0` est ajoutée à la pile pour permettre de marquer les endroits où on veut pouvoir revenir: ceci consiste à marquer toutes les variables présentes avant comme non assignées (changer leur valeur n'est pas nécessaire). Pour vous aider, une implémentation élémentaire d'une telle pile vous est donnée.

Modifiez `src/arrays.ml` afin d'utiliser des tableaux pour les affectations partielles, en utilisant la seconde approche expliquée ci-dessus. Pour tester la correction dans un premier temps vous pouvez regarder les traces sur des exemples simples, vous pouvez ensuite exécuter la commande `make PROVER=./arrays test` pour des tests plus complets. Celle-ci va également comparer votre performance avec celle de `minisat`, ce ratio est d'environ 600 dans notre cas.

Vous devez préciser (1) comment vos structures représentent l'affectation partielle et (2) quels sont les invariants/pré-conditions/post-conditions qui sont vérifiés par votre pile de modifications au cours des appels récursifs à la fonction `dpll`

2.2. Deux littéraux surveillés

Deux idées fondamentales ont permis des gains de performance impressionnants dans les solveurs SAT. La première est l'apprentissage de clause par les conflits (CDCL): elle est purement logique et très intéressante, mais en dehors de la portée de ce projet. La deuxième est les **deux littéraux surveillés** (two watched literals en anglais): il s'agit d'une amélioration algorthmique de la structure de données utilisée pour représenter les clauses, permettant une propagation unitaire plus efficace.

Un solveur DPLL passe la majorité de son temps à propager des littéraux, il est donc important d'optimiser cette étape. La propagation unitaire nécessite d'itérer sur les clauses qui sont unitaires étant donné l'affectation actuelle. Une observation clé est que, puisqu'une clause devient unitaire lorsque tous ses littéraux sauf un sont affectés à faux, il suffit de surveiller deux des littéraux d'une clause (par convention on placera les deux littéraux surveillés en début de la clause) pour détecter ceci, en maintenant l'invariant suivant:

Dans une clause non satisfaite, les deux littéraux surveillés ne sont pas affectés.

Ainsi, lorsqu'un littéral `L` devient faux, on propage de la façon suivante:

- Il n'y a pas besoin de considérer les clauses où `L` n'est pas surveillé. En effet, ces clauses ont au moins deux autres littéraux non assignés: leurs littéraux surveillés.
- On n'a aussi pas besoin de considérer les clauses satisfaites, i.e. contenant un littéral affecté à vrai. Ceci inclut les clauses contenant la négation de `L`, surveillée ou non.
- Si une clause non satisfaite où `L` est surveillé n'a qu'un autre littéral non affecté, il s'agit nécessairement de son autre littéral surveillé, et on peut le propager. Si deux telles propagations sont contradictoires, on a un conflit.
- Si une clause où `L` est surveillé n'est pas unitaire, elle possède donc un autre littéral `L'` non assigné à surveiller à la place. On maintient donc l'invariant en changeant l'ordre des littéraux de la clause, remplaçant `L` par `L'`.

Plus de détails, et des exemples peuvent être trouvés dans ce papier ou celui-ci.

Un aspect important de cette technique est que puisque le choix des littéraux surveillés est arbitraire (tant qu'il respecte l'invariant), il n'y a pas besoin d'annuler les changements de littéraux surveillés lors du backtracking. Ainsi, on peut utiliser une structure de données mutable efficace en organisant les clauses par littéraux surveillés, sans aucun coût lors du backtracking. En pratique, on associe à chaque littéral une liste doublement chaînée de clauses où le littéral est surveillé, pour pouvoir insérer et supprimer une clause en temps constant.

Adaptez `src/arrays.ml` en `src/twl.ml` en implémentant cette technique. Vous devriez remarquer un autre gain substantiel de performances, même si votre programme devrait encore être environ 200 fois plus lent que `minisat`, comme indiqué par `make test`.

2.3. Bonus

Vous êtes libres d'aller plus loin, mais le nombre de points bonus accordés sera très limité, il est donc inutile de trop en faire. Vous pouvez ainsi essayer :

- Des heuristiques comme DLIS ou Jeroslow-Wang.
- Faire un prétraitement de la formule, en éliminant les litéraux purs, les clauses tautologiques…
- Détecter automatiquement certaines symétries entre variables pour réduire l'espace de recherche (plus difficile).

Rendu 1 —— Dimanche 10 mai 23h59

Envoyer vos fichers par email sous la forme d'une archive `nom_prénom.tar.gz` contenant l'intégralité de vos fichers ainsi qu'un readme expliquant les choix que vous avez effectués et les difficultés que vous avez rencontrées.

Le rendu est le Dimanche 10 mai 23h59.