Projet DPLL

Maena Quemener

May 9, 2022

1 Introduction

Dans le cadre du cours "Projet Logique" de la L3 informatique de l'ENS Paris-Saclay, nous avons eu à implémenter un SAT-solver reposant sur l'algorithme DPLL en OCaml, puis à l'utiliser pour résoudre des sudokus. J'ai en plus rajouté la possibilité de résoudre le problème "Unequal" du site https://www.chiark.greenend. org.uk/ sgtatham/puzzles/.

2 Implémentation naïve de DPLL et optimisation

2.1 Implémentation naïve

Dans un premier, j'ai implémenté l'algorithme naïf à savoir :

- Si la CNF est vide, elle est satisfiable. On renvoie donc une liste d'assignation de variables vide. Si elle comporte une clause vide, elle est insatisfiable, on renvoie `None`. On vérifie ces deux critères.
- On regarde toutes les clauses à un littéral de la CNF, celles-ci imposent la valuation de leur variable.

  Si cette assignation impose des conflits (des singletons contenant des variables opposées), on retourne `None`, sinon on nettoie la CNF selon le principe suivant, pour chaque variable assignée :

  1. On élimine toutes les clauses comportant la variable, car elles sont satisfaites.
  2. On élimine, dans les autres clauses, tous les littéraux correspondant à l'opposé de la valuation de la variable.
- On vérifie de nouveau que la CNF n'est pas vide. Si elle l'est, on renvoie les variables qu'on vient d'assigner, sinon on choisit une variable encore présente aléatoirement.

  1. On évalue positivement la variable choisie, on nettoie la CNF sur le même principe que décrit plus haut, puis on rappelle l'algorithme récursivement sur la nouvelle CNF. Si la fonction renvoie une liste de variables, cela veut dire que la nouvelle CNF est satisfiable. On rajoute donc la variable qu'on vient de valuer et on retourne cette liste. Sinon on effectue l'étape 2.
  2. On évalue négativement la variable. De même si la fonction renvoie une liste. Si elle renvoie `None`, alors cette CNF ne peut pas être satisfiable. On renvoie donc `None`.

Avec cette version de l'algorithme, les résultats des tests, avec un `TIMEOUT=30`, j'obtiens les scores suivants :

Wrong	1
Pass	20
Timeout	42
Fail	0

En particulier, DPLL arrive à traiter 14 (WRONG inclu) tests OK et 7 tests KO. Ce qui ne semble pas étonnant car actuellement, pour montrer qu'une CNF est insatisfiable, il faut parcourir récursivement la CNF pour tout valuation de variable qu'on effectue.

2.2 Mémoïsation

Le parcours récursif des valuations m'a fait penser à implémenter de la mémoïsation. Si on obtient une sous-CNF SAT, on n'explore pas l'autre branche et on remonte directement, il n'y a donc pas besoin de les stocker. En revanche, si on obtient une sous-CNF UNSAT, on va backtracker, et tester d'autres valuations ensuite. Il est donc possible de retomber plusieurs fois sur une même sous-CNF.
Ainsi, à chaque fois qu'une CNF est UNSAT, je la stocke dans un set (j'ai pour cela rajouté un module `Memois` dans `ast.mli` pour pouvoir les stocker). Cela me permet, à chaque nouvelle entrée dans `solve`, de tester si on l'a déjà étudié cette CNF et donc si elle est UNSAT, ce qui permet beaucoup de calculs en moins.

Comme attendu, cette optimisation a énormément amélioré la rapidité de mon algorithme pour les tests KO, mais n'a rien changé pour les tests OK. Les nouveaux résultats pour un `TIMEOUT=30` sont:

Wrong	1
Pass	33
Timeout	29
Fail	0

Dont 14 tests OK (ceux-ci ne changent pas) et 20 tests KO (à savoir, 13 de plus) parmi les tests qui passent. En particulier, les tests `dubois` qui ne passaient pas avec la première version, passent tous ici en moins d'une seconde.

Pour un `TIMEOUT=120`, j'obtiens :

Wrong	2
Pass	38
Timeout	23
Fail	0

2.3 Autres tentatives d'amélioration

J'ai ensuite tenté plusieurs petites améliorations : vérifier si une variable apparaît valuée de la même manière dans chaque clause et faire un `while` sur l'étape de simplification des singletons.

Pour la première amélioration, j'ai récupéré toutes les variables dans un `Set` et j'ai regardé pour chaque variable `x` et chaque clause `c` si "`x` apparaît dans `c` ou `-x` n'apparaît pas dans `c`". Je travaille avec des sets pour effectuer ces opérations plus rapidement.
Cependant, cette opération a au contraire ralenti les tests, en particulier les tests KO. J'ai cependant laissé les fonctions correspondantes en commentaire.

En revanche la deuxième amélioration fait beaucoup gagner en efficacité, j'obtiens les résultats suivants pour un `TIMEOUT=30`:

Wrong	2
Pass	37
Timeout	24
Fail	0

A savoir seulement un de moins que sans cette amélioration, avec un `TIMEOUT=120` !
Cependant, avec un `TIMEOUT=120`, on obtient les mêmes résultats que précédemment. Le `TIMEOUT` minimal pour résoudre ce nombre de tests est de 50.

Note : Après discussion avec mes camarades, nous obtenons les même 2 tests OK WRONG.

3 Résolution de sudokus

La deuxième partie du projet consistait à modéliser la résolution d'un sudoku par une formule CNF, et ensuite la donner à notre DPLL pour qu'il le résolve.

3.1 Modélisation d'une grille de sudoku

J'ai d'abord fait un modèle qui permettait de modéliser n'importe quelle grille de sudoku sur lequel nous n'avions aucune indication.
Chaque case (i, j) du sudoku est encodé par 9 variables, une variable $x_{i,j,k}$ pour chaque chiffre $1 \leq k \leq 9$ qu'elle peut contenir. $x_{i,j,k}$ est valuée positivement si la case (i, j) contient k, négativement sinon. On a donc 721 variables au total.

Note : Pour pouvoir représenter les variables comme un un numéro dans mon programme, j'utilise la bijection suivante : $(i, j, k) \in [\![0, 8]\!] \times [\![0, 8]\!] \times [\![1, 9]\!] \to 81i + 9j + k$.

J'ai ensuite construit une CNF qui impose ces deux critères :

- **Chaque ligne, colonne, carré doit contenir au moins une case de chaque chiffre.** Pour chaque ligne, colonne ou carré, cela est représenté par 9 clauses : une pour chaque couleur. Chaque clause contient 9 littéraux positifs, un littéral pour chaque case, indiquant qu'au moins une des cases vaut ce numéro. Cela fait un total de 243 clauses.

  Exemple: La clause indiquant que la ligne 1 doit contenir un 5 est : $x_{1,1,5} \vee x_{1,2,5} \vee x_{1,3,5} \vee x_{1,4,5} \vee x_{1,5,5} \vee x_{1,6,5} \vee x_{1,7,5} \vee x_{1,8,5} \vee x_{1,9,5}$.

- **Chaque case ne peut contenir qu'un unique chiffre.** Cela est représenté, pour chaque case, par un ensemble de clauses à deux littéraux négatifs. Pour chaque paire de chiffre entre 1 et 9, on indique qu'on ne peut pas avoir les deux dans la même case, ce qui assure l'unicité du remplissage. Cela fait 36 clauses pour chaque case, et donc 2916 clauses au total.

Ces critères assurent un remplissage correct du sudoku.

3.2 Adaptation à un sudoku partiellement rempli

Cette CNF est générique, et il faut donc l'adapter à chaque grille donnée en entrée. Pour cela, on utilise le fait que connaître le chiffre d'une case permettra de valuer toutes les variables suivantes :

- La variable correspondant à la case et au chiffre est valuée positivement. Toutes celles correspondant à la même case et un chiffre différent le sont négativement.

- Toutes les variables correspondant à une case appartenant à la même ligne, colonne ou carré et à ce chiffre sont valuées négativement, car on ne peut pas avoir deux fois le même chiffre.

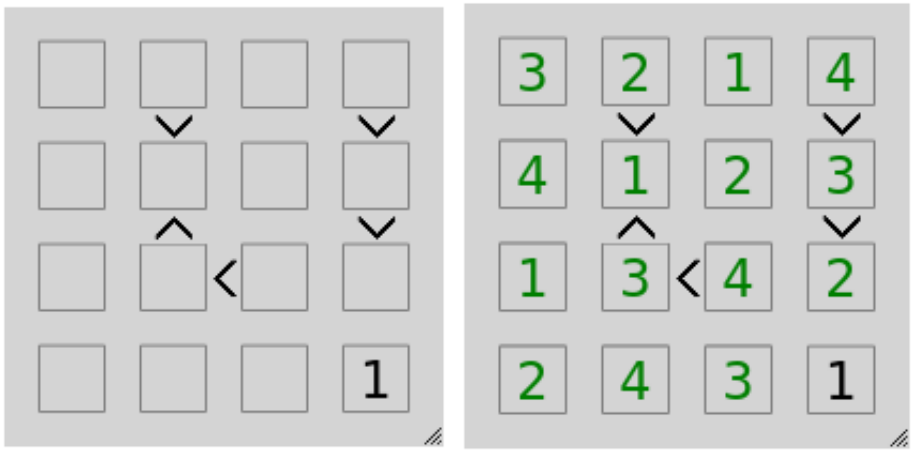
On nettoie selon le principe décrit plus haut la CNF avec cette valuation, ce qui permet de se ramener à une CNF entre 100 et 200 variables, dont généralement beaucoup de singletons, ce qui est très simple pour notre DPLL à résoudre.

3.3 Résultats

Mon algorithme résout les 1000 sudokus donnés, chacun en moins de 1 sec. En particulier, le TIMEOUT minimal pour résoudre les 50 premiers sudokus (sur mon ordinateur personnel, moins puissant que ceux de l'ENS) est de 0.3 secondes.

4 Résolution du jeu unequal

Le jeu Unequal consiste en une grille 4×4 sur laquelle certains chiffres sont indiqués, ainsi que des inégalités. Le but est de remplir chaque case avec des chiffres entre 1 et 4, de manière à avoir un unique chiffre de chaque sur chaque ligne, colonne et que les inégalités soient respectées. Exemple de grille, et sa solution :



4.1 Modélisation d'une grille

Similairement au sudoku, on modélise la grille par un chaîne de caractère qui représente chaque case, à la différence qu'on rajoute ensuite 12 caractères pour les inégalités horizontales, puis 12 encores pour les verticales.

On représente une case vide par 0, sinon on indique le numéro, on indique $<$ par 1 et $>$ par 2, pour les inégalités verticales, $\wedge$ est représenté par 1 et $\vee$ par 2.

Exemple sur la grille donnée : 0000 0000 0000 0001 pour le remplissage; 000 000 010 000 pour les inégalités horizontales; 0202 0102 0000 pour les inégalités verticales; le tout concaténé et sans espace.

Pour la sortie, on renvoie juste une chaîne qui correspond au remplissage.
Exemple sur la grille donnée : 3214 4123 1342 2431.

4.2 Modélisation du problème

Pour cela, on code les cases par des variables sur le même principe que le sudoku. De même pour les clauses imposant l'unicité d'un chiffre sur une ligne ou une colonne.

En revanche, on rajoute des clauses de taille 2 pour indique tous les couples de remplissage non possible quand on a une inégalité (dans l'exemple, on rajoute une clause pour dire qu'on ne peut pas avoir $x_{0,1} = 1, x_{1,1} = 2$ entre autres).

On nettoie notre CNF ainsi obtenue avec les assignations de variables qu'on connaît déjà grâce aux chiffres placés, puis on la donne à notre DPLL.

4.3 Exécution et résultats

Pour l'exécution, j'ai modifié le `MakeFile` de manière à pouvoir tester mon programme, j'ai rajouté un `unequal.mli` et une fonction `handle_unequal` dans le `main.ml`.

Pour tester la résolution de problèmes unequal, j'ai rajouté 20 tests dans le fichier `unequal` qui représentent tous une grille différente. Les tests s'exécutent sur le même principe que les sudokus, avec `make unequal`. Par défaut, 10 problèmes vont être testés, à partir du test 0. On peut adapter ça en modifiant les variables `NUNEQUAL` et `UNEQUALSTART`.

Mon programme réussit les 20 tests, et très rapidement.