De Jesus Jethro **Le Bretton Tanguy** **Zekri Ramdane**

Rapport de projet sur le solver SAT

M1 Informatique – Université Paris Descartes 2018-2019

Le problème de satisfiabilité booléenne (SAT) est un problème de décision, c'est-à-dire, étant donné une formule de logique propositionnelle, est-ce que ce problème est satisfiable ? Nous devions implémenter un solver SAT qui devait répondre à cette question et, si la formule est satisfiable, donner un modèle de la formule. Dans notre cas, les différentes formules données sont des CNF (conjonctions de clauses).

Nous avons décidé d'utiliser le C++ comme langage de programmation car c'est le langage que nous sommes en train d'apprendre durant ce semestre et c'est un bon compromis entre le C et le Java. En effet, le C++ est rapide à l'exécution, plus complet que le C et nous n'avions pas besoin de toutes les fonctionnalités de l'orienté objet (uniquement de manipuler des vecteurs).

Choix d'implémentation

Notre code est séparé en 3 fichiers sources : le fichier qui contient les algorithmes de backtracking et de simplification, le fichier qui permet la lecture d'un fichier DIMACS et le fichier qui contient la méthode main.

De la même façon que pour le fichier dimacs fourni en entrée, un littéral est représenté sous la forme d'un entier (int). Ainsi pour obtenir la négation de ce littéral, il suffit de le multiplier par -1.

Une clause étant définie comme une disjonction de littéraux, celle-ci sera représentée comme un vecteur d'entier vector<int>. L'utilisation de la classe vector nous est utile dans de nombreux cas, notamment lorsque l'on souhaite supprimer un littéral ou une clause.

Une formule CNF est représentée comme un vector<vector<int>> dans notre algorithme. La CNF étant une conjonction de clauses, la représentation sous la forme d'un vecteur de clauses s'est faite de façon naturelle.

Lecture du fichier DIMACS

La lecture du fichier est assez basique. On vérifie d'abord l'extension du fichier fourni en paramètre (un fichier DIMACS possède l'extension ***.cnf***). On teste ensuite si le fichier existe et s'il est lisible. A partir de là, on lit la première ligne qui contient des informations telles que le nombre de variables ainsi que le nombre de clauses du fichier. Une fois le nombre de variables obtenu, on crée le vecteur contenant toutes ces variables. Ce vecteur était fourni vide par référence à la fonction de telle sorte à ce que les modifications soient visibles à l'extérieur de la fonction et que les autres fonctions puissent l'utiliser. On parcourt le reste du fichier en ajoutant chaque variable à la clause actuelle tant qu'on ne rencontre pas un 0 dans le fichier. Dès que le curseur atteint un 0, cela signifie qu'on arrive à la fin de la clause et on passe à la clause suivante. On retourne ensuite le vector<vector<int>> qui est la liste des clauses.

Backtracking

Le ***Backtracking*** est une technique algorithmique servant à résoudre des problèmes récursivement en essayant de trouver une solution incrémentalement, en supprimant les solutions qui ne répondent pas à la contrainte du problème. Ici, nous avons utilisé un vecteur 2D appelé ***cnf*** pour représenter notre formule. Une clause est un vecteur, ***cnf[i]***. Un littéral est un élément de ce dernier, ***cnf[i][j]***. Nous avons également utilisé un vecteur d'entiers pour regrouper les littéraux de notre CNF appelé ***var***. Finalement, un vecteur d'entiers qui stockent les ***models***.

Avec ***l'algorithme backtracking***, nous avons choisi le dernier élément de notre vecteur avec ***var.back()*** et l'avons nommé ***v***. Nous avons assigné la valeur 1 à ***v***. Ensuite, nous devons enlever ***v*** de notre vecteur ***var***. Il faut par la suite simplifier ***cnf*** en utilisant ***simplifyCnf(cnf, v)*** et nommé cela ***simplified*** afin de déterminer si l'on peut obtenir une formule satisfaisable ou pas.

Si ***simplified.empty()*** retourne *true*, nous avons trouvé un modèle et la formule sous forme de CNF est satisfaisable. Si ce n'est pas le cas, nous devons vérifier si ***simplified*** contient une ou des clause(s) vide(s) en utilisant ***emptyClause(simplified)***. S'il n'y a pas de clause(s) vide(s) dans ***simplified***, nous ajoutons ***v*** au ***model*** et nous calculons récursivement ***l'algorithme backtracking*** sur ***simplified***. Si au contraire il y a une ou des clause(s) vide(s), nous devons simplifier ***cnf*** avec ***¬v*** ce qui assigne 0 à ***v*** ce qu'on appellera ***simplified_second*** afin d'empêcher la formule d'être insatisfiable.

Si ***simplified_second.empty()*** retourne *true*, nous avons trouvé un modèle et la formule sous forme de CNF est satisfaisable. Sinon, nous devons vérifier si ***simplified_second*** contient une ou des clause(s) vide(s) en utilisant ***emptyClause(simplified_second)***. ***emptyClause(vector<vector<int>> cnf)*** est une fonction qui retourne *true* si ***cnf*** contient une clause vide, une vecteur vide, ou *false* si l'on ne trouve pas de clause vide dans ***cnf***. Si la fonction retourne vrai, la formule sous forme de CNF est insatisfiable. Sinon, nous ajoutons ***¬v*** au ***model*** et calculons récursivement ***l'algorithme backtracking*** sur ***simplified_second***.

Efficacité

Sur les instances de test fournies par l'enseignant, notre algorithme fonctionne dans 88% des cas.