Rapport Projet 2

Kawisorn Kamtue, Emirhan Gürpınar

Nous avons programmé **en C++**, en utilisant beaucoup d'éléments de **C**.

Organisation du code

Le code est structuré de la manière suivante :

preproc

Le préprocessus pour transformer l'entrée en forme postfix

On lit l'entrée standard si un nom de fichier n'est pas donné pendant l'appel de l'exécutable **resol**, sinon on lit le fichier dont le nom est donné. En utilisant une pile on transforme l'entrée sous un autre format qui est postfix et qui a la lettre i au début de chaque entier. Les priorités sont respectés, dans l'ordre décroissant: parenthèses ( ), et /\, ou \/, ou exclusive X, implication =>, equivalence <=>, négations ~ -. On fait la distinction entre la négation d'un variable - et la négation d'une formule ~.

tseitin

Le transformé de Tseitin étant optionnel, on crée le fichier cnf à partir de la forme postfix.

On lit les variables et tant que l'on lit les operateurs on les applique aux variables. La structure de donnée que l'on utilise est l'arbre binaire. Le fils droite est toujours NULL pour l'opérateur négation. On transforme tout d'abord la formule en une formule équivalente ou les opérateurs AND ne sont pas au dessous d'un opérateur que AND et les opérateurs négation ne sont pas en dessus des autres opérateurs. Ensuite on applique le transformée de Tseitin, récursivement.

sat

Le solveur sat prenant en entrée le fichier cnf, le clause learning est optionnel. Ce fichier contient l'algorithme DPLL et CDCL.

Dans les rendus précedents, on crée les vecteurs pour stocker chaque clause. On se trouve souvent dans le problème "segmentation fault".

Dans cette version de SAT solver, on utilise la matrice de taille fixe 1000x1000 donc mat[i] represent la clause i.
mat[i][j] vaut 1 si la clause i contient j, -1 si elle contient -j et 0 sinon. Si la clause contient à la fois j et -j, on supprime cette clause dans
le pretraitement (dans main.cpp). De plus, mat[i][0] = la valeur de la clause i au niveau courant donc on n'a pas besoin de réevaluer clause i si elle
est déjà mis à vrai ou faux. On garde le niveau ou la clause i est mis à vrai ou false.

Pour chaque variable, on garde son valeur (1 si vrai, -1 si faux et 0 si inconnu). On garde aussi le niveau(date) ou sa valeur est donnée vrai ou faux.

L'algorithme general: on fait la dept-first-search jusqu'on rencontre le conflit et on backtrack. Les nodes sont les variables paris.

Pour choisir le nouvelle variable, il y a 3 options: la prochaine variable inconnu dans l'ordre croissant (default), le variable qui apparaît le plus dans les clauses de taille minimale parmi les clauses vivantes (option MOMS), et la variable qui rend le plus de clauses satisfaites parmi les clauses vivantes.

Pour la deduction, on utilise deux fonctions dans la deduction: unit propagation et unique polarite. Cette deux fonctions retourne le variable deduit si on peut déduire quelque chose, 0 sinon. On repète ces fonctions tant qu'ils ne retournent pas 0.

Pour backtrack, on met tout les variable et les clauses dont ses niveau de décision sont supérieur à niveau courant(variable globale). Pour DPLL, on décremente le niveau courant par 1.

Lorsqu'on active l'option CL, le parcours est differemment. Tant que on ne trouve pas le conflit et on ne satisfie pas tous les clause, on parie toujours "vrai" sur la prochaine variable. Si on trouve le conflit, on fait l'analyse du conflit. On ajoute nouvelle clauses au cours de l'analyse de conflit. Dans le mode interactif, on peut cree le graphe du conflit et affiche le dernier clause ajoutée (UIP choisir). On remet le niveau courant au niveau determiné par l'analyse du conflit. Si on trouve le conflit au niveau 0, on retourn UNSATISFIABLE.
\item main:\newline
Le programme fondamental qui lit l'entrée de l'utilisateur et qui fait les appels nécessaires aux fichier mentionnés ci-dessus.
\end{itemize}

Bibliography

www.slideshare.net/sakai/how-a-cdcl-sat-solver-works
en.wikipedia.org/wiki/Tseytin_transformation