Solveur SAT Automne 2011

FOURURE Damien 10806634 & DUHAMEL Arnaud 10502972

30 janvier 2012

1 Définition des différents types et structures

1.1 La Formule conjonctive

```
typedef vector<clause *> formule_conj;
```

Comme cela est indiqué dans le sujet, nous avons défini une formule conjonctive comme étant un tableau de pointeurs vers des clauses. Le pointeur est utile en particulier lors de l'indexation des clauses (cf. partie 1.3).

1.1.1 La Clause

```
typedef pair<bool,vector<int> > clause;
```

Pour pouvoir déclarer une formule conjonctive comme étant un tableau de clauses, il faut définir ce qu'est une clause. Dans un premier temps, une clause était un simple tableau d'entier. Mais pour optimiser le code (lors de la partie 2.3) nous avons modifié les clauses pour qu'elles soient une paire comprenant un booléen et un tableau d'entier. Ainsi, quand une clause est évaluée à vrai, le booléen est mis à vrai, et par conséquent il n'est plus nécessaire de la parcourir à nouveau.
Attention : cela nécessite de garder en mémoire les clauses évaluées à vrai pour faire du backtracking.

1.2 L'interprétation

```
typedef vector<int> interpretation;
```

Une interprétation est une fonction qui associe à une variable une valeur. Ici les valeurs ne sont pas booléennes mais 1,−1 ou 0 comme indiqué dans le sujet. Du coup un simple tableau suffit pour stocker la valeurs des variables.
Attention : Le fait que la numérotation des variables commence à 1 peut induire en erreur, puisque les cases d'un tableau commencent à 0. De ce fait, pour obtenir la valeur de la variable i il est nécessaire de faire : interprétation[i-1].

1.3 L'indexation

```
struct indexation {
        vector< pair<list<clause*>,int> > positif;
        vector< pair<list<clause*>,int> > negatif;
};
```

Le but de l'indexation est de faire correspondre à chaque **littéral** une liste de clause. Le principe est donc le même que pour l'interprétation, sauf que l'on utilisera un tableau de pointeurs sur clause. De plus, on veut séparer les positifs et les négatifs, on utilise donc deux tableaux. Enfin, pour pouvoir gérer les variables monotones, nous avons du rajouter un entier, qui correspond au nombre de clauses qui ne sont pas évaluées à vrai, et dans lesquelles le littéral apparait (cf. 2.2.2).

2 Codage des fonctions

2.1 Un solveur naif

2.1.1 Numérotation des variables

La fonction numérote est très facile. La seule subtilité concerne l'insertion dans un map. En effet, l'insertion ne se fait que si la clé (ici le nom de la variable) n'existe pas encore. Il n'est donc pas nécessaire de tester l'existence de la variable dans le map.

2.1.2 Traduction en forme conjonctive

Nous avons choisi la traduction linéaire équisatisfiable car elle est plus rapide et plus évidente à faire une fois le cours assimilé. Cependant, cette traduction supporte moins bien certaines optimisations.
Avant de traduire une formule en forme conjonctive, nous avons codé deux types de fonctions : les fonctions **et**, qui retournent une formule de forme conjonctive, et les fonctions **ou**, qui retournent des clauses.

- **Les fonctions ou :** ces fonctions permettent de créer des clauses. Nous avions commencé avec les deux prototypes suivant :

  ```
  extern clause * ou(int valeur);
  extern clause * ou(int arg1, clause * arg2);
  ```

  La première fonction créée une nouvelle clause (avec un new clause()) et appelle la deuxième qui a pour effet d'introduire le int passé en paramètre dans la clause.
  Par la suite, nous nous sommes rendu compte que lors de la traduction linéaire, les clauses créées n'avaient jamais plus de trois littéraux, et par conséquent, pour plus de lisibilité nous avons créé les fonctions **ou** suivantes :

  ```
  extern clause * ou(int val1, int val2);
  extern clause * ou(int val1, int val2, int val3);
  ```

  Enfin, par souci de flexibilité, nous avons créé une dernière fonction **ou** qui prend en paramètre deux clauses et qui les fusionnent. Cette fonction est inutile dans notre cas, mais aurait pu l'être si nous avions choisi la transformation en forme conjonctive directe.
- **Les fonctions et :** ces fonctions permettent de créer des formules conjonctives. Nous n'avons eu besoin que des deux fonctions suivantes :

  ```
  extern formule_conj * et(clause *  arg1, formule_conj * arg2);
  extern formule_conj * et(formule_conj * arg1, formule_conj * arg2);
  ```

  La première insère la clause dans une formule conjonctive, la deuxième fusionne deux formules conjonctives.

Une fois ces fonctions écrites, la seule difficulté a été de comprendre à quoi correspondait le triplet indiqué dans le sujet. Ensuite il a suffit de suivre le cours.

2.1.3 Exploration de l'espace de rechercher

Aucune difficulté pour cette partie là, il a suffit de suivre le sujet du Tp. En revanche, ces fonctions ont été largement modifiées par la suite. Par contre, le sujet suggérait de commencer par la variables 0 puis d'avancer en appellant récursivement avec $var + 1$. Nous avons choisi de partir de la dernière variable, puis d'appeler récursivement avec $var - 1$ car l'ordre selon lequel nous construisons la formule conjonctive implique que la dernière variable introduite fini toujours toute seule dans une clause. Il est donc plus logique de commencer par elle.

2.2 Propagation et retour arrière

2.2.1 Couper l'arbre de recherche

Dans la première version de la fonction cherche, nous appelions récursivement la fonction si nous n'avions pas attribué toutes les variables. Ici, l'idée est de tester la formule conjonctive avant, et d'appeler récursivement sur la variable suivante que si cette formule renvoie la valeur indéterminée (= 0).

2.2.2 Indexation des clauses

Le principe de cette fonction est très simple : on fait une première boucle pour parcourir toutes les clauses. Puis dans une seconde boucle, on parcourt tous les littéraux de la clause. Enfin, pour chaque littéral trouvé, on insère dans la case correspondante de l'indexation (cf. 1.3) le pointeur de la clause où se trouve ce dernier.

L'indexation a ensuite été modifiée pour prendre en compte les variables monotones (cf. 2.3.1)

2.2.3 Propagation Unaire

Pour la propagation unaire, nous avons du modifier la fonction clause_est_satisfaite, afin que dans le cas où la clause est indéterminée et ne possède qu'un seul littéral indéterminé, elle retourne (par le biais d'un passage de référence) le littéral indéterminé. Ensuite, il suffit de faire comme c'est indiqué dans le sujet du Tp.

2.2.4 Retour arrière

Le retour arrière consiste juste à remettre la valeur de toutes les variables dans la liste *déduite* à zéro puis réinitialiser la liste.

2.3 Optimisation

2.3.1 Variable monotone

Pour introduire les variables monotones, nous avons dû modifier la définition d'une clause pour y introduire un booléen (cf. 1.1.1) qui permet de ne pas réévaluer une clause que l'on sait à vrai. Ensuite, nous avons rajouté dans l'indexation un entier pour compter les clauses qui ne sont pas évaluées à vrai (cf. 1.3). Ensuite nous avons codé une fonction monotone qui prend une clause en paramètre et décrémente tous les compteurs des littéraux qu'elle contient. Ensuite, si un compteur tombe à zéro, cela signifie que l'opposé du littéral devient monotone et donc on peut mettre à vrai toutes les clauses qui le contient (en rappelant récursivement la fonction monotone).
Il faut aussi penser à garder en mémoire toutes les clauses mise à vrai pour pouvoir les remettre à faux dans la fonction cherche (c'est la même logique que pour les variables déduites).

3 Test

Pour tester l'exactitude de notre programme, nous avons fait le script shell suivant :

```
I=0;
while test "$I" != "$1"
do A=$(python genere_pb.py -s -t 1000 | ./SAT) && echo "$I" "$A"
I=$(expr "$I" + 1)
done
```

Le script consiste en une boucle qui génère une formule qu'on pipe sur l'entrée du programme.
Nous avons lancer deux fois le script, une fois avec des formules satisfiables et une fois avec des formules insatisfiables, pour qu'il boucle 10000 fois dans les deux cas. Dans tous les cas notre programme nous à retourné la bonne valeur. On peut donc considérer qu'il est opérationnel.

4 Problème rencontré

Après avoir coder les variables monotones, nous avons voulu tester l'efficacité de l'optimisation. Nous avons donc écris le script shell suivant :

```
A=$(python genere_pb.py -s -t 3000)
echo "$A" | time ./SAT
echo "$A" | time ./SAT_NM
```

Ce shell génère une formule, puis la test avec un SAT qui gère les variables monotones puis un SAT_NM qui ne les gères pas. Le résultat ne fut pas concluant étant donné que le SAT gérant les monotones est généralement plus lent.

Après une longue réflexion, nous avons compris d'où vennait le problème :
Nous avons choisi la transformation en forme conjonctive linéaire or cette transformation ne créé que des clauses d'au maximum trois littéraux. Gérer les variables monotones revient à mettre à vrai certaines clauses sans avoir à les évaluer. Or avec seulement trois littéraux dans une clause, le gain de temps est inférieur au cout de traitement des monotones.
En conclusion, cette optimisation aurait été beaucoup plus efficace si nous avions choisi la transformation en forme conjonctive directe.