L3-MI / PROG
TP#8

2023-W47/48

Instructions de rendu

Ce TP fait l'objet d'un rendu *individuel ou en binôme*. Vous devez envoyer votre travail sous la forme d'une archive à :

philip.scales@univ-grenoble-alpes.fr ;

au plus tard le vendredi 2023-12-08 à 18 :00. Celle-ci doit contenir :

- Votre programme répondant aux questions ci-dessous. Il doit être possible de facilement exécuter toutes les expériences permettant de répondre aux questions.
- Un Makefile et les instructions permettant de l'utiliser.
- Un compte-rendu détaillé au format PDF qui décrit vos choix de conception et vos réponses aux questions.

Remarque : Le sujet de ce TP étant assez long, il n'est pas nécessaire de traiter toutes les questions afin d'obtenir une note maximale.

Résolution exhaustive de 3SAT avec bitslicing

Le but de cet exercice est d'implémenter un *SAT-solveur* par recherche exhaustive, qui utilise le *bitslicing* afin d'être plus efficace qu'une implémentation naïve du même algorithme.

Un petit nombre de fonctions utilitaires est déjà fourni, et la structure du programme est en grande partie imposée. Tout ceci est fourni *via* le fichier `bessats.c` (contenu dans l'archive https://membres-ljk.imag.fr/Pierre.Karpman/pc2023_tp8.tar.bz2), et décrit ci-dessous.

Type `slice`

Le programme utilise un type abstrait `slice` pour la représentation des *slices* du bitslicing. Vous devrez instancier ce type avec des entiers de 64 bits, via le type `uint64_t`, et avec des vecteurs de 256 bits, via le type `__m256i` (dans ce cas, il sera nécessaire d'utiliser les instructions du jeu d'extension AVX2). Pour chaque instanciation, il est nécessaire d'implémenter chacune des fonctions de l'interface (imposée) pour le type `slice`, documentées dans le fichier source fourni avec le sujet.

Représentation des instances 3SAT et de leurs solutions

Dans le programme, la représentation d'une instance 3SAT se fait de la façon suivante :

— une variable de type `int8_t` est utilisée pour stocker le nombre de variables.

— un type `struct` est utilisé pour représenter une clause :

```
typedef struct clause
{
    int8_t a;
    int8_t b;
    int8_t c;
} clause;
```

Les valeurs absolues des membres `a`, `b`, `c` d'une structure `clause` indiquent les variables apparaissant dans la clause, et leurs signes indiquent si les littéraux correspondent à la variable « nue » ou niée par $\neg$.
REMARQUE : Ce choix de représentation impose que les variables soient indexées à partir de 1 (et non 0, ce qui empêcherait de distinguer les littéraux z_0 et $\neg z_0$).
EXEMPLE : La clause $z_1 \vee z_7 \vee \neg z_3$ peut être représentée par la structure dont les membres `a`, `b`, `c` valent 1, 7 et -3. L'ordre des littéraux dans l'écriture de la clause n'étant pas important, on peut aussi par exemple représenter cette même clause par la structure dont les membres `a`, `b`, `c` valent 7, 1 et -3.

— l'ensemble des clauses formant une instance 3SAT est stocké comme un tableau ou dans un espace mémoire contiguë accessible via un pointeur de type `clause *`.

Dans le programme, la représentation d'une solution à une instance 3SAT à n variables se fait simplement par un tableau d'entiers de type `int8_t` (ou dans un espace mémoire contiguë accessible via un pointeur de type `int8_t *`), où l'élément à l'index i est nul si la variable z_{i+1} est à 0 (ou « faux ») dans la solution, et à n'importe quelle valeur non nulle si cette même variable est à 1 (ou « vrai »).
EXEMPLE : La solution formée par l'affectation $z_1 = 0$, $z_2 = 1$, $z_3 = 1$ peut être représentée par le tableau dont les éléments aux indexes 0, 1, et 2 valent respectivement 0, 2, et 3.

Le fichier source fourni avec le sujet donne également trois fonctions utilitaires :

— `void gen_rand_system(int8_t nvar, size_t nclause, clause cl[nclause])`

génère de façon pseudo-aléatoire un système à `nvar` variables et `nclause` clauses.

— `void print_system(int8_t nvar, size_t nclause, clause cl[nclause])`

affiche sur la sortie standard un système représenté comme ci-dessus ; l'affichage du système est fait au format DIMACS.

— `void print_sol(int8_t nvar, size_t nclause, clause cl[nclause],`
`↪ int8_t sol[nvar], bool res)`

si `res` est à `true`, affiche `SAT` sur la sortie standard, et la solution représentée comme ci-dessus ; l'affichage de la solution est fait au format DIMACS. Sinon, affiche `UNSAT`.

EXEMPLES : Une fois un solveur implémenté, les fonctions ci-dessus peuvent être utilisées pour produire des sorties similaires à celles fournies dans les fichiers `exsat.cnf` et `exunsat.cnf`.

Travail préparatoire

Q.1 : Écrivez les fonctions (qui ne sont pas déjà implémentées) de l'interface `slice` dans le cas où le type est instancié avec des entiers de type `uint64_t`. Ces fonctions

correspondent dans le fichier source fourni au cas où le symbole SL256 est non défini, c'est à dire à l'intérieur des blocs suivant les directives `#ifndef SL256`.

L'écriture de certaines de ces fonctions peut être grandement facilitée par l'emploi d'instructions seulement disponibles avec les jeux d'extension BMI1, BMI2, POPCNT... Il est pour cela conseillé d'utiliser les *intrinsèques* proposées par les compilateurs, et par exemple documentées à : https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#othertechs=BMI1,BMI2,POPCNT.

Résolution exhaustive naïve : `_tiny_solve`

Q.2 : Écrivez la fonction :

```
bool _tiny_solve(int8_t nvar, size_t nclause, clause cl[nclause], int8_t
↪  sol[nvar])
```

qui tente de trouver une solution à l'instance 3SAT fournie en argument.

Cette fonction doit implémenter l'algorithme de recherche exhaustive *sans utiliser de bitslicing*.

Il est vivement conseillé d'utiliser un code combinatoire pour énumérer les affectations possibles pour les variables.

À titre indicatif, cette fonction ne devrait pas faire plus d'une trentaine de lignes.

Q.3 :

1. Testez la correction de votre fonction `_tiny_solve`. Vous pouvez pour cela comparer les résultats de votre fonction sur des instances aléatoires avec ceux d'un solveur externe, par exemple cryptominisat [1]. Alternativement, vous pouvez vérifier de petites instances « à la main ».
2. Testez les performances de votre fonction `_tiny_solve`. À titre indicatif, vous devriez être capable de résoudre des instances en 32 variables et 144 clauses (similaires à celles données en exemple) en au plus quelques minutes dans le pire cas.

Résolution exhaustive bitslicée : `bs_solve`

Q.4 : Écrivez la fonction :

```
bool bs_solve(int8_t nvar, size_t nclause, clause cl[nclause], int8_t
↪  sol[nvar])
```

qui tente de trouver une solution à l'instance 3SAT fournie en argument.

Cette fonction doit implémenter l'algorithme de recherche exhaustive *en utilisant du bitslicing* (sauf dans les cas où le nombre de variables est inférieur à `sl_lsz`, le logarithme de la taille en bit d'une variable `slice` ; dans ce cas la fonction peut utiliser `_tiny_solve` en sous-routine).

Les opérations sur les *slices* doivent se faire uniquement en utilisant le type `slice` et ses fonctions associées.

Le bitslicing doit s'effectuer en utilisant l'approche décrite en cours.

Il est vivement conseillé d'écrire cette fonction en prenant comme base votre fonction `_tiny_solve`, et en adaptant ce qui est nécessaire ; à titre indicatif, cette fonction doit faire nettement moins qu'une petite centaine de lignes.

1. https://github.com/msoos/cryptominisat

Q.5 :

1. Testez la correction de votre fonction `bs_solve` en comparant ses résultats sur des instances aléatoires avec ceux de `_tiny_solve`.
2. Testez les performances de votre fonction `bs_solve`. À titre indicatif, vous devriez être capable de résoudre des instances en 32 variables et 144 clauses en au plus quelques secondes dans le pire cas, et des instances de 36 variables et 180 clauses en quelques dizaines de secondes dans le pire cas.

Q.6 : Écrivez les fonctions (qui ne sont pas déjà implémentées) de l'interface `slice` dans le cas où le type est instancié avec des vecteurs de type `__m256i`. Ces fonctions correspondent dans le fichier source fourni au cas où le symbole SL256 est défini, c'est à dire au cas *`#else`* des directives *`#ifndef SL256`*.

Ceci nécessite l'emploi d'instructions seulement disponibles avec les jeux d'extension AVX et AVX2. Il est pour cela conseillé d'utiliser les *intrinsèques* proposées par les compilateurs, et par exemple documentées à : https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX_ALL.

Q.7 : Vous devez maintenant pouvoir choisir entre les deux instanciations de `slice` rien qu'en définissant ou non le symbole SL256.

1. Testez la correction de votre fonction `bs_solve` avec des *slices* de 256 bits en comparant ses résultats sur des instances aléatoires avec ceux de `_tiny_solve`.
2. Testez les performances de votre fonction `bs_solve` avec des *slices* de 256 bits. À titre indicatif, vous devriez être capable de résoudre des instances en 40 variables et 200 clauses en au plus quelques minutes dans le pire cas.

Résolution de #SAT

Le problème #SAT est similaire au problème SAT, et a pour seule différence que pour une instance satisfiable, un algorithme pour #SAT doit retourner le nombre (supérieur ou égal à 1) d'affectations distinctes pour lesquelles la formule est satisfaite.

Q.8 : Modifiez votre fonction `bs_solve` de façon à ce que si le symbole `SHARP` est défini, elle renvoie le nombre (éventuellement nul) d'affectations satisfaisantes ; vous pouvez pour cela modifier la signature de `bs_solve`.

Remarque : Il peut être utile ici d'utiliser la fonction `popcount` du type `slice`.

Q.9 : Testez la correction de votre fonction en comparant ses résultats sur des instances aléatoires avec ceux d'un solveur externe, par exemple cryptominisat.

Q.10 : Utilisez votre fonction pour étudier l'évolution du nombre de solutions en fonction du nombre de clauses relativement au nombre de variables. Vous pouvez par exemple chercher à reproduire une figure similaire à la Figure 1 (pas nécessairement avec le même nombre de variables).

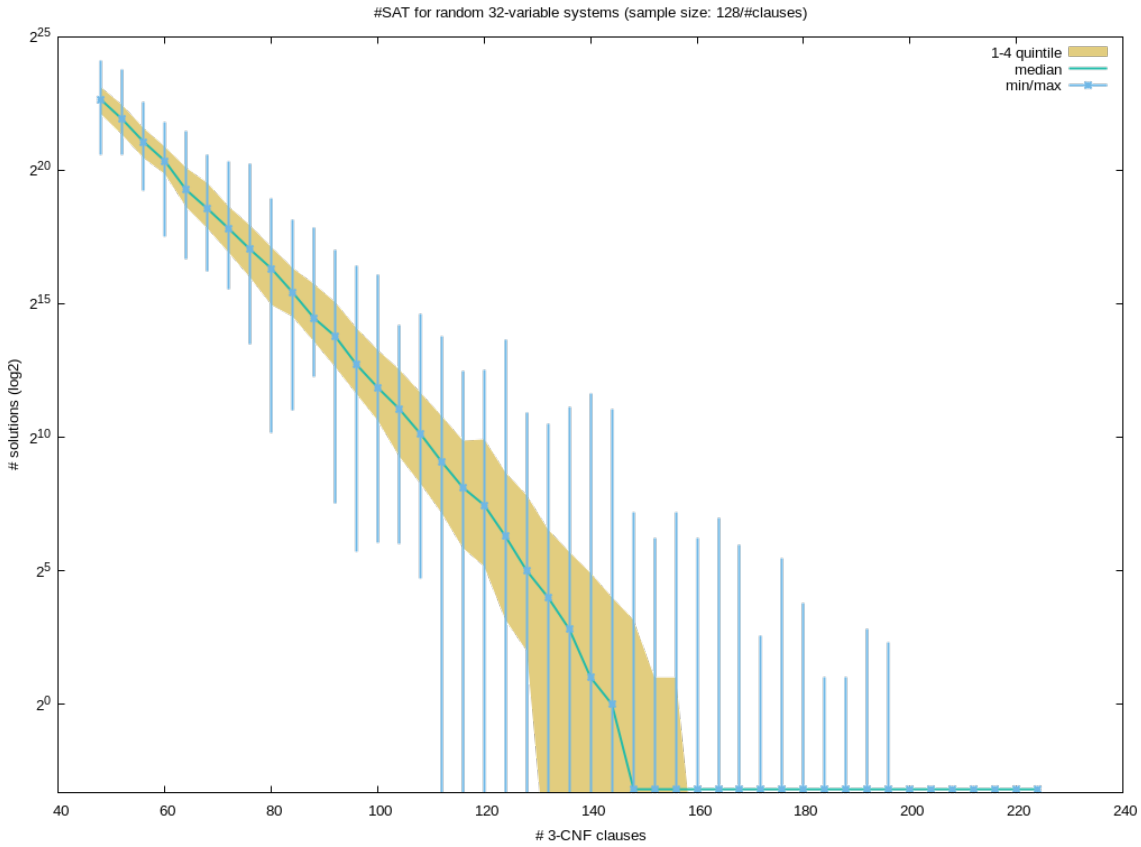

FIGURE 1 – Statistiques pour #3SAT sur des instances aléatoires à 32 variables