TP #28 — Résolution de 3-SAT par recherche exhaustive

Le but de ce sujet est d'implémenter en C la résolution d'instances du problème 3-SAT par une recherche exhaustive. On commence par une implémentation naïve de cette approche, que l'on va ensuite légèrement améliorer. Dans une seconde partie on exploitera le parallélisme au niveau bit des instructions des processeurs usuels pour accélérer la recherche par un facteur constant notable (par exemple 64, voire plus). En guise d'ouverture, on proposera d'adapter les fonctions implémentées (de façon relativement immédiates) à la variante de *comptage* #3-SAT, et (de façon nettement moins immédiate) à un algorithme asymptotiquement meilleur que la recherche exhaustive.

Problème 3-SAT

Une instance du problème 3-SAT est donnée par une formule propositionnelle φ sur un certain ensemble de variables $\mathcal{V}$ en 3-CNF : c'est à dire que φ est une conjonction de clauses chacune égale à une disjonction de trois littéraux (donnés chacun par une variable propositionnelle $v \in \mathcal{V}$ ou sa négation). Résoudre l'instance φ consiste à décider si elle est satisfiable ou non, c'est à dire décider si elle admet un modèle ou non, et éventuellement renvoyer un modèle dans le cas d'une réponse positive. Autrement dit, on cherche à décider s'il existe une valuation (ou affectation) σ des variables de $\mathcal{V}$ telle que $\sigma \vDash \varphi$ (ou $[\![\varphi]\!]_\sigma = 1$).

Résolution exhaustive. Une résolution d'une instance 3-SAT par recherche exhaustive suit l'algorithme extrêmement simple suivant : on énumère les $2^{\#\mathcal{V}}$ valuations σ de $\mathcal{V}$ dans un ordre quelconque, et pour chacune on teste si $\sigma \vDash \varphi$. Si l'on trouve un tel modèle alors on répond `SAT` (pour « satisfiable ») et éventuellement une représentation de σ, et sinon l'on répond `UNSAT` (pour « insatisfiable »). Soit $e(\varphi)$ le coût (éventuellement amorti) d'une évaluation de φ et $n := \#\mathcal{V}$ le nombre de variables, cet algorithme a un coût pire cas $O(e(\varphi) \times 2^n)$. Si φ est de taille polynomiale en n, on a $e(\varphi) = O(n^c)$ pour une certaine constante c, et une recherche exhaustive a alors un coût $\widetilde{O}(2^n)$, où la notation $\widetilde{O}$ « cache » ici les facteurs polynomiaux en n.

Représentation informatique des formules en 3-CNF & leurs modèles

Dans tous le sujet on prendra $\mathcal{V} = [\![1, 63]\!]$, ce qui n'est pas une limitation étant donné que nous ne seront pas capable de résoudre des instances en plus de quelques dizaines de variables en temps raisonnable. On peut alors représenter le littéral v par l'entier de type `int8_t` égal à v, et le littéral $\neg v$ par celui égal à $-v$.

C'est ce choix de représentation qui motive le fait que nous avons exclu 0 de $\mathcal{V}$, puisqu'alors on ne pourrait pas distinguer une représentation de 0 et $\neg 0$. La seconde borne à 63 simplifiera quant à elle nettement l'implémentation d'une énumération de toutes les valuations.

On représentera une clause disjonctive $a \vee b \vee c$ de littéraux a, b, c par le type :

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

où les champs `.a`, `.b`, `.c` représentent les littéraux a, b, c de la manière décrite ci-dessus.

Une formule sous 3-CNF sera représentée par un tableau de `struct clause`, ou généralement un pointeur `struct clause *` pointant vers le début d'une zone mémoire contenant des valeurs de type `struct clause` stockées de façon contiguë.

Enfin, un modèle sera simplement représenté par un tableau de `int8_t`, ou généralement un pointeur `int8_t *` pointant vers le début etc., dont l'élément à l'indice i vaut `0` pour une affectation de la variable $i+1$ à « faux », et n'importe quelle valeur non nulle pour une affectation à « vrai ».

Fichier de démarrage

On fournit un fichier `tp28.c` qui qui implémente trois fonctions utilitaires :

— `void gen_rand_system (size_t nvar, size_t nclause, clause cl[nclause])`
génère pseudo-aléatoirement une formule 3-CNF de `nvar` variables et `nclause` clauses, en utilisant une fonction `randbits` *que vous devez remplacer par une fonction de type de retour* `uint32_t` *ou* `uint64_t` *de génération de nombre pseudo-aléatoire*. Si elle est disponible sur votre système, on conseille d'utiliser pour cela `arc4random`.
Dans le cas d'un grand nombre de clauses relativement au nombre de variables, il se peut que `gen_rand_system` produise des formules possédant plusieurs fois la même clause, ce qui n'est pas très pertinent. Cependant, la distribution suivie par les formules générées est suffisamment « raisonnable » pour notre usage.

— `void print_system (size_t nvar, size_t nclause, clause cl[nclause])`
affiche sur la sortie standard une formule représentée comme ci-dessus. Cet affichage se fait au format « DIMACS », qui est un format assez primitif mais couramment employé par les « vrais » *SAT solvers* (comme cryptominisat) pour décrire une instance 3-SAT.

— `void print_sol (size_t nvar, int8_t sol[nvar], bool res)`
affiche UNSAT si `res` vaut `false`, et sinon SAT suivi d'une représentation au format DIMACS de l'affectation contenue dans `sol`.

Opérateurs bit-à-bit

Les fonctions à écrire dans Ce sujet bénéficient avantageusement (et pour certaines, nécessitent) l'utilisation des opérateurs « bit-à-bit » fournis par le langage C (pas vraiment au programme, mais tolérés). Ceux-ci sont par exemple décrits à la §10.3 du poly de C.

Implémentation naïve d'une résolution exhaustive

1. Écrivez une fonction C de signature :
   `void int2vec(uint64_t x, size_t nvar, int8_t s[nvar])`
   qui pour `nvar` au plus 63 modifie `s[i]` en la valeur du i-ème bit de `x` en partant du bit de poids faible.
   Par exemple, après un appel `int2vec(0xA, 4, s)`, l'on aura que `s[0] == s[2] == 0` et `s[1] == s[3] = 1`.
   Remarque. Cette fonction s'écrit aisément en utilisant les opérateurs de calcul du quotient & reste dans la division entière, mais vous pouvez aussi à leur place utiliser les opérateurs `>>` et `&`.
2. Écrivez une fonction C de signature :
   ```
   bool tiny_solve
   (size_t nvar, size_t nclause, clause cl[nclause],
   int8_t sol[nvar])
   ```
   qui résout l'instance de problème 3-SAT représentée par `cl`, en implémentant naïvement une résolution exhaustive : il suffit d'utiliser la fonction `int2vec` et un compteur pour générer toutes les affectations possibles à considérer, et de tester pour chacune si elle est un modèle de la formule en argument.
3. Testez, par exemple sur de petites instances générées aléatoirement que vous résoudrez à la main, ou en comparant vos résultats avec ceux d'un *SAT solver* externe.
   (N'oubliez pas l'existence des fonctions `print_system` et `print_sol`.)
4. Donnez le coût pire cas de `tiny_solve` en fonction de `nvar` et `nclause`.

On peut légèrement améliorer les performances de `tiny_solve` en diminuant le coût de calcul des affectations. Pour cela on utilise un *code combinatoire*, qui permet d'énumérer les 2^n valeurs d'un mot binaire de n bits de sorte que deux valeurs successives diffèrent exactement d'un bit. On illustre cette idée sur un exemple avec $n = 3$, où l'on énumère les mots du code (pour un certain code)

ainsi que les action à effectuer sur un tableau représentant une affectation, afin que celle-ci prenne successivement toutes les valeurs possibles :

```
— /* 000 */ int8_t s[3] = {0, 0, 0}
— /* 001 */ s[0] ^= 1
— /* 011 */ s[1] ^= 1
— /* 010 */ s[0] ^= 1
— /* 110 */ s[2] ^= 1
— /* 111 */ s[0] ^= 1
— /* 101 */ s[1] ^= 1
— /* 100 */ s[0] ^= 1
```

L'utilisation de l'opérateur « ^ » de *ou exclusif* (en anglais : *exclusive or*, ou *XOR*) simplifie ici les changements de `s` : on a `0 ^ 1 == 1` et `1 ^ 1 == 0`, ce qui fait qu'il n'est pas nécessaire de retenir (ou lire) la valeur actuelle de l'élément de `s` à l'indice à changer : connaître l'indice seul est suffisant.
Pour rendre cette approche vraiment intéressante, il est nécessaire d'être capable de déterminer efficacement la suite des indices devant être modifiés. C'est en fait assez facile pour le code (infini) dont est extrait l'exemple ci-dessus : l'indice à changer à l'étape i est simplement donné par la valuation 2-adique de i, ou plus prosaïquement le nombre de bits de poids faible à zéro dans l'écriture de i en base deux (son *number of trailing zeros* (`ntz`), ou *trailing zero count* (`tzcnt`)).
Il existe de nombreux algorithmes astucieux pour calculer ce dernier, mais de nos jours le plus simple est généralement d'utiliser l'instruction dédiée à cette opération fournie par votre processeur. En C sur architecture `x86`, cette instruction s'accède typiquement à travers l'*intrinsèque* `_tzcnt_u64`. Celle-ci fournit une fonction de signature (compatible avec) `uint64_t _tzcnt_u64(uint64_t a)` qui renvoie la valuation 2-adique de l'entier représenté par `a`. Son utilisation nécessite **à la fois** l'inclusion du fichier d'entête `immintrin.h` **et** de compiler le programme avec une option `-mbmi` ; dans le cas contraire, la compilation peut échouer ou non en fonction du compilateur, mais même quand celle-ci n'échoue pas le code compilé risque d'être moins efficace.
Remarque. Nous ne rentrerons pas dans le détail de la nature des intrinsèques, mais il est bon de retenir qu'elles ne font pas partie du langage C lui-même (et donc ne sont surtout pas au programme). En conséquence, un programme utilisant une intrinsèque sera généralement peu *portable*, et sa sémantique dépendra notamment du compilateur utilisé.

5. Écrivez une fonction C de signature :
   `uint64_t ntz(uint64_t x)`
   qui renvoie la valuation 2-adique de son argument. *Si possible*, utilisez pour cela l'intrinsèque `_tzcnt_u64` ou équivalente (et sinon, un algorithme naïf fera l'affaire).
6. Testez.

7. Écrivez une fonction C de signature :
   ```
   bool tiny_solve2
   (size_t nvar, size_t nclause, clause cl[nclause],
   int8_t, sol[nvar])
   ```
   qui adapte votre fonction `tiny_solve` en utilisant une énumération efficace des affectations.
8. Testez, y compris les performances. À titre indicatif, en compilant votre programme avec l'option d'optimisation `-O2`, 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.

Accélération par *bitslicing*

Les opérateurs bit-à-bit logiques (et les instructions auxquelles ils correspondent typiquement sur les processeurs) ont la caractéristique d'effectuer un grand nombre de calculs *en parallèle*. Par exemple, l'opérateur `&` avec des opérandes de 64 bits calcule 64 opérations de ET logique *indépendantes* en parallèle. On peut exploiter cela pour accélérer une résolution exhaustive de 3-SAT d'un facteur égal à la longueur (en bit) des opérandes, en utilisant une technique dite de *bitslicing*.
Dans notre cas, cette technique s'instancie de la façon suivante : on utilise des mots binaires v_i d'une certaine longueur w (par exemple 64), un pour chaque variable propositionnelle, et l'on définit la j-ème *tranche* (*slice*) de ces mots comme l'ensemble des j-ème bits de chacun des mots v_i. Si toutes les tranches sont distinctes, alors utiliser des opérateurs bit-à-bit pour évaluer une formule logique sur les v_is permet d'évaluer celle-ci *simultanément* sur toutes les affectations correspondant aux valeurs des tranches.
On illustre cette technique sur un exemple jouet avec $w = 8 = 2^3$, qui va nous permettre d'évaluer *en une seule fois* une certaine formule :

$$\texttt{MAJ}(a, b, c) = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

(qui n'est pas en 3-CNF, mais cela ne change rien) pour *toutes les affectations possibles* de ses trois variables (dit autrement, on va en une seule évaluation parallèle calculer l'intégralité de la table de vérité de `MAJ`) :

— On définit :
  ```
  uint8_t a = 0b10101010;
  uint8_t b = 0b11001100;
  uint8_t c = 0b11110000;
  ```
  où l'on constate que la i-ème tranche (lue « verticalement », en partant de la droite, ordonnée de a (moins significatif) à c) est une représentation de l'écriture de i en base deux sur trois bits.
— On calcule :

```
uint8_t c1 = a & b;   // 0b10001000;
uint8_t c2 = a & c;   // 0b10100000;
uint8_t c3 = b & c;   // 0b11000000;
uint8_t d1 = c1 | c2; // 0b10101000;
uint8_t d2 = d1 | c3; // 0b11101000;
```

et le résultat est donné par les tranches de `d2`.

— Puisque `d2` n'est ni égal à `0`, ni à `255`, on peut déduire que `MAJ` est satisfiable mais n'est pas une tautologie. Les indices des bits où il vaut `1` indiquent les tranches de `a`, `b`, `c` qui sont des modèles de `MAJ` ; par exemple puisque son 3-ème bit vaut un l'on a que $a = 1$, $b = 1$, $c = 0$ en est un modèle.

Pour accélérer une résolution exhaustive de 3-SAT par *bitslicing*, on propose l'approche suivante : soit w la longueur des mots utilisés et ℓ_w sont logarithme en base deux (par exemple $w = 64$ et $\ell_w = 6$) :

— Soit n le nombre de variables de l'instance à résoudre, si $n < \ell_w$ on la résout (très vite) avec `tiny_solve`.

— Sinon, on choisit arbitrairement ℓ_w variables et l'on initialise les mots les représentants de façon à ce que leurs w tranches représentent les $w = 2^{\ell_w}$ affectations possibles pour ces variables. On initialise les $n - \ell_w$ autres mots tout à zéro, et l'on énumère les $2^{n-\ell_w}$ valeurs qu'ils peuvent prendre, *en étant ou bien tout à zéro ou bien tout à un* (par exemple en utilisant un code combinatoire, et l'opérateur bit-à-bit « ~ » de négation) : l'idée est que pour chaque évaluation, les affectations de ces $n - \ell_w$ variables soient *identiques*, tandis qu'elles prennent toutes les valeurs possibles sur les ℓ_w autres variables.
On a alors bien accéléré la recherche exhaustive par un facteur w.

Le fichier fourni `tp28.c` définit un type `word`, instancié par un `uint64_t`, ainsi que des valeurs en lecture seule `w_sz`, `w_lsz`, et des signatures de fonctions opérant sur les `words`. Vos fonctions *bitslicées* devront être écrites en utilisant exclusivement le type `word` pour représenter les mots. Ceci permettra le cas échéant d'obtenir des versions avec d'autres tailles de mot (par ex. 256) en changeant uniquement l'implémentation du type `word`.

9. Implémentez les fonctions opérant sur les `words` dont les signatures sont fournies.

10. Écrivez une fonction C de signature :

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

de mêmes spécifications que `tiny_solve`, et qui utilise l'approche par *bitslicing* décrite ci-dessus pour accélérer la recherche exhaustive.
Remarque. À part son initialisation, cette fonction est très similaire à `tiny_solve2`.

11. Testez, y compris les performances. Vous devriez être capable de résoudre des instances en 32 variables et 144 clauses en au plus quelques secondes dans le pire cas.

Problème #3-SAT

Il existe une variante de *comptage* du problème 3-SAT (et du problème SAT en général), notée #3-SAT (prononcer *sharp* 3-SAT (!)) qui consiste à dénombrer les modèles d'une formule. Une résolution de 3-SAT par recherche exhaustive peut naturellement être adaptée à #3-SAT, puisqu'il suffit de compter les modèles jusqu'au dernier plutôt que de (possiblement) interrompre l'énumération des affectations dès que l'on en a trouvé un.
Pour maintenir le caractère modulaire de la résolution par *bitslicing*, il est pratique d'introduire une fonction de signature `uint64_t popcount(word x)` qui renvoie le nombre de bits à un dans `x`.

12. Implémentez une telle fonction. *Si possible*, utilisez pour cela l'intrinsèque `_mm_popcnt_u64` (ou similaire).
13. Écrivez une fonction `sharpsolve` qui adapte votre fonction `solve` à la résolution de #3-SAT.

Transition de phase. L'évolution avec le nombre de clauses (de l'espérance) du nombre de solutions d'une instance 3-SAT à nombre de variable fixe (générée aléatoirement suivant une distribution uniforme) subit un phénomène de *transition de phase* : il existe une valeur α (conjecturée environ égale à 4.25) telle qu'une instance à n variables est presque sûrement satisfiable quand le nombre de clauses est $< \alpha n$ et presque sûrement insatisfiable sinon, et ceci a également un impact sur le nombre de solutions.
Vous pouvez essayer d'utiliser votre implémentation de `sharpsolve` pour observer cela expérimentalement, en traçant de jolis (?) graphes comme celui-ci de la Figure 1.

De plus grands mots

Si votre processeur le permet, vous pouvez instancier le type `word` pour une plus grande taille, par exemple 256 bits, voire 512. Sous Linux, le *pseudofichier* `/proc/cpuinfo` vous permet de connaître (notamment) les extensions de jeu d'instruction présentes sur votre processeur (indiquées par leurs *flags*). Si (par exemple) les *flags* `avx` et `avx2` sont présents, vous devriez pouvoir utiliser les intrinsèques `_mm256_...` (par exemple `_mm256_and_si256`) pour instancier le type `word` avec le « type » `__m256i` (ou équivalent).

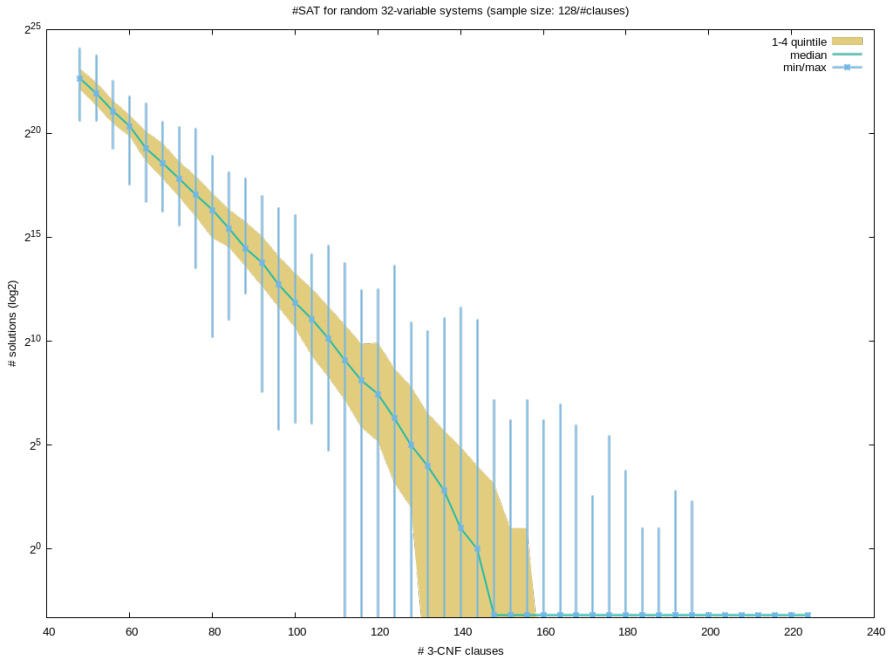

FIGURE 1 – Nombre de solutions pour des instances 3-SAT uniformes en 32 variables

Un meilleur algorithme

En pratique, les *SAT solvers* permettent de résoudre de nombreuses instances 3-SAT (mais typiquement *pas* #3-SAT) très largement hors de portée d'une résolution par recherche exhaustive : leur performance « moyenne » est *heuristiquement* bien supérieure à une telle approche.
On connaît également des algorithmes dont le coût **pire-cas** est significativement meilleur qu'une recherche exhaustive. Un tel algorithme très simple utilise de petites marches aléatoires autour d'affectations aléatoires. On peut montrer que pour une instance à n variables, $\widetilde{O}(\frac{4}{3}^n)$ tentatives sont suffisantes pour trouver un modèle (s'il en existe un) avec forte probabilité. Ceci donne naturellement un algorithme probabiliste de type « Monte-Carlo » du même coût.

Cet algorithme s'implémente assez facilement, et peut aussi être accéléré par une technique de *bitslicing* (un peu différente de celle employée ci-dessus pour la recherche exhaustive).