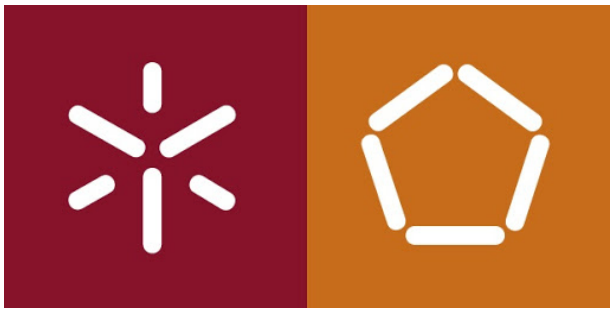
SAT solving - Questões para Avaliação



Luís Ribeiro (A85954)
Mestrado em Engenharia Informática
Universidade do Minho

1 | Matriz

Considere o seguinte programa C sobre inteiros:

```
for(i=1; i<=3; i++)
    for(j=1; j<=3; j++)
        M[i][j] = i+j;
```

1.1 Alínea 1

> Escreva um programa C, constituído apenas por uma sequência de atribuições, que seja equivalente.

Para isto, declarei uma matriz de inteiros desta maneira: `int M[4][4];`. Como i e j estão declarados no intervalo de 1 a 3, a matriz deve ser 4 por 4 por considerar o índice 0 como primeira posição.
As atribuições foram feitas através do desenrolamento dos ciclos. Tendo as atribuições inciais $i = j = 1$, o primeiro ciclo fica desta maneira,

```
/* Primeiro ciclo i=1 */
int i=1;int j=1; /* Declarações iniciais */
M[i][j] = i+j;
j=j+1;
M[i][j] = i+j;
j=j+1;
M[i][j] = i+j;
```

Dentro do ficheiro `matriz-e1.c` podemos encontrar todas estas declarações e o restantes ciclos.

1.2 Alínea 2

> Faça a codificação lógica do programa que definiu na alínea anterior, num ficheiro em formato `SMT-LIBv2`.

Para a resolução desta alínea foi preciso ter em conta a familiarização com as teorias e lógicas `SMT-LIB` sobre *arrays* funcionais, com extensionalidade. Para isso foi usado a lógica `QF_AUFLIA`, estudada em aula.
Para a matriz definida, sendo ela considerada um *array* de *array* de inteiros, considerei esta declaração,

```
(declare-const matriz (Array Int (Array Int Int)))
```

Esta matriz definida em cima, será a matriz final definida, visto que para ser feito os *stores* é necessário a declaração de 3 matrizes e 9 *arrays* (3 para cada matriz). Desta maneira temos,

```
; Inicialização, para atribuir os primeiros valores. Matriz e Array auxiliares vazios.
(declare-const m0 (Array Int (Array Int Int)))
(declare-const m00 (Array Int Int))

; Declaração das matrizes e sub-matrizes (arrays).
(declare-const m1 (Array Int (Array Int Int)))
(declare-const m11 (Array Int Int))
(declare-const m12 (Array Int Int))
(declare-const m13 (Array Int Int))

(declare-const m2 (Array Int (Array Int Int)))
(declare-const m21 (Array Int Int))
(declare-const m22 (Array Int Int))
(declare-const m23 (Array Int Int))

(declare-const matriz (Array Int (Array Int Int)))
(declare-const m31 (Array Int Int))
(declare-const m32 (Array Int Int))
(declare-const m33 (Array Int Int))
```

Para simular as atribuições dos valores i e j declarados em C, também é necessário a declaração das seguintes variáveis.

```
; Declaração dos i,j
(declare-const i Int)
(declare-const j Int)
(declare-const i1 Int)
(declare-const j1 Int)
(declare-const i2 Int)
(declare-const j2 Int)
```

As atribuições da matriz foram feitas através de `asserts` de igualdade entre `store's` da matriz e valores constantes i e j que têm o valor inicial $= 1$.

Por exemplo, o primeiro ciclo $i = 1$, é feito da seguinte maneira,

```
;; Primeiro ciclo i = 1
  ; Inical i e j
  (assert (= i 1))
  (assert (= j 1))
  (assert (= m11 (store m00 j (+ i j))))
```

```
(assert (= j1 (+ j 1)))
(assert (= m12 (store m11 j1 (+ i j1))))

(assert (= j2 (+ j1 1)))
(assert (= m13 (store m12 j2 (+ i j2))))

; Primeira matriz definida
(assert (= m1 (store m0 i m13)))
```

A declaração final da `matriz` será esta,

```
; Matriz final definida
(assert (= matriz (store m2 i2 m33)))
```

As restantes declarações encontram-se explícitas dentro do ficheiro `matriz-e2n3.smt2`.

1.3 Alínea 3

> Tendo por base a codificação lógica que fez do programa, use um `SMT` solver para estabelecer, no final da execução do programa, a validade das propriedades que se seguem:

Para as propriedades definidas entre a alínea (b) e (e), foi necessário a criação de novas variáveis/constantes.

```
; Para as restantes alíneas, de modo a não haver repetição de restrições.
(declare-const ii Int)
(declare-const jj Int)
(declare-const aa Int)
(declare-const bb Int)

(assert (and (>= ii 1) (<= ii 3)))
(assert (and (>= jj 1) (<= jj 3)))
(assert (and (>= aa 1) (<= aa 3)))
(assert (and (>= bb 1) (<= bb 3)))
```

Note que,
Para provar que as propriedades são verdade, temos que demonstrar que cada propriedade é uma consequência lógica das condições acima. Para isso temos que negar cada propriedade e verificar se o problema é **NÃO satisfazível**.

1.3.1 Alínea (a)

> Se $i = j$ então $M[i][j] \neq 3$.

A negação desta propriedade é definida desta maneira,

Negação da propriedade:

```
(assert (not (=> (= i0 j0) (not (= (select (select matriz i0) j0) 3)))))
```

Para esta propriedade, é preciso declarar um novo i e j que poderão tomar qualquer valor.

```
(declare-const i0 Int)
(declare-const j0 Int)
```

Resultado: O problema é satisfazível (SAT), logo esta propriedade não é uma consequência lógica das condições acima. O contra exemplo apresentado, é tal onde $i0 = j0 = 8$ e $M[8][8] = 3$. Isto acontece porque não impusemos valores restritos a $i0$ e $j0$, e a matriz apenas tá declarada para valores de i, j até 3.

1.3.2 Alínea (b)

Para quaisquer i e j entre 1 e 3, $M[i][j] = M[j][i]$.

A negação desta propriedade é definida desta maneira,
Negação da propriedade:

```
(assert (not (= (select (select matriz ii) jj) (select (select matriz jj) ii))))
```

Resultado: O problema é NÃO satisfazível (UNSAT), logo a propriedade é verdadeira, por ser uma consequência lógica às condições anteriores.

1.3.3 Alínea (c)

Para quaisquer i e j entre 1 e 3, se $i < j$ então $M[i][j] < 6$.

A negação desta propriedade é definida desta maneira,
Negação da propriedade:

```
(assert (not (=> (< ii jj) (< (select (select matriz ii) jj) 6))))
```

Resultado: O problema é NÃO satisfazível (UNSAT), logo a propriedade é verdadeira, por ser uma consequência lógica às condições anteriores.

1.3.4 Alínea (d)

Para quaisquer i, a e b entre 1 e 3, se $a > b$ então $M[i][a] > M[i][b]$.

A negação desta propriedade é definida desta maneira,
Negação da propriedade:

```
(assert (not (=> (> aa bb)
(> (select (select matriz ii) aa) (select (select matriz ii) bb)))))
```

Resultado: O problema é NÃO satisfazível (UNSAT), logo a propriedade é verdadeira, por ser uma consequência lógica às condições anteriores.

1.3.5 Alínea (e)

> Para quaisquer i e j entre 1 e 3, $M[i][j] + M[i+1][j+1] = M[i+1][j] + M[i][j+1]$.

A negação desta propriedade é definida desta maneira,

Negação da propriedade:

```
(assert (not (= (+ (select (select matriz ii) jj)
(select (select matriz (+ ii 1)) (+ jj 1))) (+
(select (select matriz (+ ii 1)) jj) (select
(select matriz ii) (+ jj 1))))))
```

Resultado: O problema é satisfazível (SAT), logo esta propriedade não é uma consequência lógica das condições acima. O contra exemplo apresentado, é tal onde $ii = 1$ e $jj = 3$, tal que $M[ii][jj] = 4$. Isto acontece porque, o valor jj é incrementado em certos cálculos ($M[i][j+1]$ e $M[i+1][j+1]$), e a matriz não está definida para $j+1 = 4$, logo vai atribuir valores a $M[1][4]$ e $M[2][4]$ de modo a satisfazer a propriedade.

2 | Puzzle Solver - Futoshiki

> Desenvolva um programa para resolver, à sua escolha, um dos seguintes puzzles, como auxílio de um `SMT solver`.

O *puzzle* escolhido foi o `Futoshiki` porque me pareceu o mais simples de implementar, visto que é muito parecido ao *Sudoku*, não tendo restrição nas sub-Matrizes definidas. No entanto, pode ter relações de pré-ordem entre posições, restringindo assim algumas atribuições de números a blocos.

Para implementação deste programa usei a linguagem **Python**, usando como auxílio a biblioteca disponível e utilizada na aula prática `z3-solver`.

Para o cálculo das restrições, é preciso saber o tamanho do tabuleiro, N. De notar que neste puzzle, ao contrário do *Sudoku*, por não haver o conceito de sub-Matriz, o valor de N lido no ficheiro de *input*, corresponderá também ao número de linhas, número de colunas e aos números inteiros de colocação. Para isto, a primeira linha do ficheiro de *input* deve ser da forma `N=X` onde X será um inteiro, à escolha do utilizador, que será atribuído a N. Então, deve ser lido inicialmente apenas esta primeira linha para que seja possível calcular as restrições de que cada número (no intervalo de 1 a N) deve estar definido em cada linha e em cada coluna. Para além disso, temos que especificar que cada posição $x_{i,j}$ (sendo i linha e j coluna), deve ter um valor correspondido entre 1 a N.

Estas restrições são feitas através da sintaxe definida pela biblioteca `z3-solver`, mencionada acima. É preciso criar uma estrutura, que será uma matriz $x[l][c]$, onde cada posição $x_{i,j}$ será declarada de forma a ser aceite pela sintaxe do `z3-solver`. A declaração de um **Solver** s também é necessário.

Após a leitura de N do ficheiro de *input* são definida as seguintes condições:

```
# restrições de bloco
for i in range(linhas):
    x[i] = {}
    for j in range(colunas):
        x[i][j] = Int('x'+str(i)+str(j))
        s.add(And(1<=x[i][j],x[i][j]<=N))

# restrições de linha
for i in range(linhas):
  s.add(Distinct([x[i][j]
        for j in range(colunas)]))
```

```
# restrições de coluna
for j in range(colunas):
  s.add(Distinct([x[i][j]
        for i in range(linhas)]))
```

Depois de calculado estas restrições, é feito um *push*, para que o resto do ficheiro de *input* seja lido, visto que este também tem a atribuição de valores a posições, tais como as relações de pré-ordem entre posições. Estas devem ter a seguinte sintaxe,

```
= (3,4) 2
< (1,2) (1,3)
```

As atribuições são feitas através de um *assert* de igualdade, depois da leitura separada de cada argumento. Sendo estes, i linha, j coluna e n valor, tendo assim, $s.add(x[i][j] == n)$. As relações de pré-ordem são feitas através de um *assert* de pré-ordem, semelhante ao anterior, no entanto ambas as posições lidas devem ser próximas, isto é, uma das posições tem que estar imediatamente acima, abaixo ou ao lado da outra.

Depois de calculados todas as restrições, e as declarações dadas como *input* terem sido adicionadas, é criado um ficheiro de *output*, onde será imprimido a resolução do tabuleiro, tal como as restrições de pré-ordem. De modo a ser possível a criação de vários *outputs* para vários *inputs*, o ficheiro de *input* deverá ter o nome começado pela *tag input*. Será criado depois, um ficheiro de *output*, com o mesmo nome do ficheiro dado, mas com a substituição da *tag* de *input*.

Exemplificando, `input-easy` → `output-easy`.

Dentro do ficheiro `futoshiki-program` podemos encontrar toda a especificação do programa feito. Alguns ficheiros de *input* foram criados com nomes `input-easy` e `input-hard`, de forma a ter 2 níveis de dificuldade do puzzle.

Um exemplo de execução do programa, recebendo o ficheiro *input* de dificuldade fácil,

```
$ ./futoshiki-program input-easy
```

Gera o seguinte *output* `output-easy`,

```
======= Solution =======
[2, 3, 4, 1]
[3, 2, 1, 4]
[4, 1, 3, 2]
[1, 4, 2, 3]

======= Pre-Orders =======
x01 < x02
x22 < x21
```