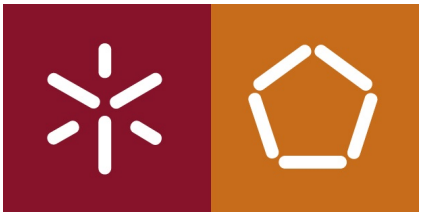

Universidade do Minho
Mestrado Integrado em Engenharia Informática
4ºano - 2º Semestre

Verificação Formal

SMT solving

a85573 - Jorge Gabriel Alves Cerqueira

12 de março de 2021

1 Matriz

A partir do seguinte código.

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

Podemos fazer *loop unrolling* e obter as seguintes instruções.

```
M[1][1] = 2;
M[1][2] = 3;
M[1][3] = 4;
M[2][1] = 3;
M[2][2] = 4;
M[2][3] = 5;
M[3][1] = 4;
M[3][2] = 5;
M[3][3] = 6;
```

Para a modelação deste loop foram usadas duas constantes, uma com a matriz inicial e outra com a matriz final.

```
(declare-const M0 (Array Int (Array Int Int)))
(declare-const M1 (Array Int (Array Int Int)))
```

Foram também criadas funções auxiliares para aceder e alterar índices na matriz de forma fácil e legível.

```
(define-fun assignM
  ((x Int) (y Int) (val Int) (m (Array Int (Array Int Int))))
  (Array Int (Array Int Int))
  (store m x (store (select m x) y val)))

(define-fun accessM
  ((x Int) (y Int) (m (Array Int (Array Int Int))))
  Int
  (select (select M1 y) x))
```

Com isto podemos facilmente simular todas as atribuições.

```
(assert (= M1
           (assignM 1 1 2
           (assignM 1 2 3
```

```
(assignM 1 3 4
(assignM 2 1 3
(assignM 2 2 4
(assignM 2 3 5
(assignM 3 1 4
(assignM 3 2 5
(assignM 3 3 6 M0))))))))))))
```

Tendo isto testar as afirmações é trivial, as constantes i e j foram previamente definidas para as equações apresentadas de seguida.

- Se $i = j$ então $M[i][j] \neq 3$.

  ```
  (assert (not (=> (= i j) (not (= 3 (accessM i j M1))))))
  ```

  O modelo acima dá *sat* logo a afirmação é falsa.

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

  ```
  (assert (and (<= 1 i) (<= i 3)))
  (assert (and (<= 1 j) (<= j 3)))
  (assert (not (= (accessM i j M1) (accessM i j M1))))
  ```

  O modelo acima dá *unsat* logo a afirmação é verdadeira.

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

  ```
  (assert (and (<= 1 i) (<= i 3)))
  (assert (and (<= 1 j) (<= j 3)))
  (assert (not (=> (< i j) (< (accessM i j M1) 6))))
  ```

  O modelo acima dá *unsat* logo a afirmação é verdadeira.

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

  ```
  (assert (and (<= 1 i) (<= i 3)))
  (declare-const a Int)
  (declare-const b Int)
  (assert (and (<= 1 a) (<= a 3)))
  (assert (and (<= 1 b) (<= b 3)))
  (assert (not (=> (> a b) (> (accessM i a M1) (accessM i b M1)))))
  ```

  O modelo acima dá *unsat* logo a afirmação é verdadeira.

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

```
(assert (and (<= 1 i) (<= i 3)))
(assert (and (<= 1 j) (<= j 3)))
(assert (not (=
              (+ (accessM i j M1) (accessM (+ i 1) (+ j 1) M1))
              (+ (accessM (+ i 1) j M1) (accessM i (+ j 1) M1)))))
```

O modelo acima dá *sat* logo a afirmação é falsa.

É de notar que as afirmações falsas acima o são devido a valores fora dos índices com valores atribuídos para o array, é um erro comum aceder a posições de memória fora do que foi alocado ou atribuído.

2 Futoshiki

Futoshiki é um jogo lógico cujo objetivo é preencher uma matriz em que cada célula precisa ser preenchida com um número entre 1 e N, - sendo N o numero de colunas da matriz - entre linhas e colunas todos os números têm que ser distintos, para além disso, cada puzzle terá regras diferentes em que células já têm valores preenchidos ou são forçadas relações de desigualdade - maior ou menor - com células adjacentes.

2.1 Exemplo de modelação

O exemplo a ser modelado será o seguinte:

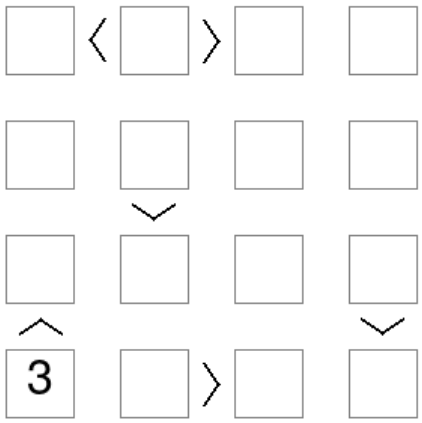

Figura 1: Exemplo de um puzzle futoshiki.

Primeiramente temos que dedicar as células, na solução encontrada cada célula terá uma constante cujo valor será um *Int* correspondente ao valor

dessa célula, iremos precisar de N^2 constantes para representar todas as células da matriz.

```
(declare-const x11 Int)
(declare-const x12 Int)
...
```

Após declaradas as constantes temos que garantir que os seus valores se mantêm dentro dos limites do puzzle, ou seja, entre 1 e N.

```
(assert (and (<= 1 x11) (<= x11 4)))
(assert (and (<= 1 x12) (<= x12 4)))
...
```

Para garantir que os valores entre as mesmas linhas e colunas são diferentes entre si podemos usar a função *distinct*.

```
; distinct values between lines
(assert (distinct x11 x12 x13 x14))
(assert (distinct x21 x22 x23 x24))
...

; distinct values between columns
(assert (distinct x11 x21 x31 x41))
(assert (distinct x12 x22 x32 x42))
...
```

Para as desigualdades podemos afirmar uma relação de superioridade entre as duas células em questão.

```
(assert (> x21 x11))
(assert (> x21 x13))
...
```

Para as células já preenchidas precisamos apenas de afirmar uma igualdade com o valor em questão.

```
(assert (= 3 x14))
```

Podemos de seguida verificar a satisfazíbilidade do modelo bem como o valor das constantes.

```
(check-sat)
(get-value (x11 x21 x31 x41 x12 x22 x32 x42
            x13 x23 x33 x43 x14 x24 x34 x44))
```

Com o modelo completo podemos agora obter a solução com um *SMT solver*.

```
> z3 futoshiki.smt2
sat
((x11 1) (x21 3) (x31 2) (x41 4)
 (x12 4) (x22 2) (x32 3) (x42 1)
 (x13 2) (x23 1) (x33 4) (x43 3)
 (x14 3) (x24 4) (x34 1) (x44 2))
```

Preenchendo a matriz podemos verificar que este resultado está de facto correto.

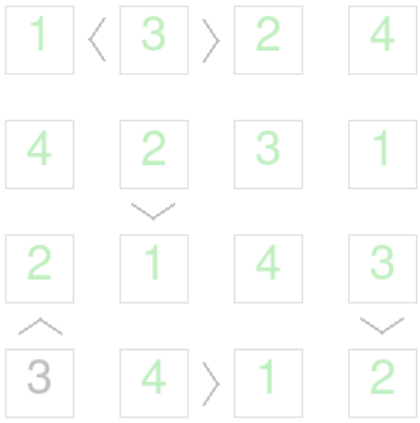

Figura 2: Puzzle futoshiki resolvido.

2.2 *Solver* desenvolvido

O *solver* foi desenvolvido em *haskell* com *stack* para gerir as bibliotecas utilizadas, esta linguagem foi escolhida pois torna o *parsing* dos ficheiros de *input* e *output* do *smt solver* relativamente trivial.
O ficheiro input é lido e a informação relevante ao problema é guardada na seguinte estrutura.

```haskell
newtype Cell = Cell { getCell :: (Int, Int) }

data Ruleset = Ruleset
  { size :: Int
  , constraints :: [(Cell, Cell)] -- Cell1 > Cell2
  , filledCells :: [(Cell, Int)]
  } deriving Show
```

Após esta estrutura o programa cria um processo com o *smt solver* e através de pipes gera e envia-lhe os comandos necessários para modelar o problema, de seguida recebe o output do *smt solver* e imprime uma solução, caso exista, para o terminal ou também, opcionalmente, para um ficheiro dado como argumento.
As seguintes regras são utilizadas para formar os puzzles:

- diferentes linhas são separadas por *newline*
- diferentes colunas são separadas por espaços
- uma célula vazia é representada com '#'
- uma célula preenchida é representada com o número correspondente à célula
- '<' denota que a célula atual é maior do que aquela à esquerda
- '>' denota que a célula atual é maior do que aquela à direita
- '^' denota que a célula atual é maior do que aquela acima
- 'v' denota que a célula atual é maior do que aquela abaixo
- comandos de comparação podem ser encadeados e pode ser adicionado um numero no seu final, ou seja uma célula representada como '<>3' é válida

Podemos agora resolver um puzzle mais complicado do que o apresentado anteriormente com a ajuda deste *solver*, a codificação será a seguinte.

```
# < # # > # # # #
# # v # # v v ^ #
# # # # # # v # #
# ^ <4 # # < # # <8
^2 9 # # # # <> # v
# # > ^v ^ 9 v < 4
# < # # > # # # <
v> ^ 8 2 ^ > # # 1
# 6 1 # # # <> # 9
```

E com o solver obtemos a solução.

```
> stack run examples/9by9.txt
Spawned solver process.
Problem is satisfiable.
6 8 2 4 7 5 9 1 3
9 7 5 1 3 8 6 4 2
8 2 3 7 4 1 5 9 6
1 3 4 9 6 7 2 5 8
2 9 6 3 1 4 8 7 5
5 1 7 6 2 9 3 8 4
3 4 9 5 8 2 1 6 7
7 5 8 2 9 6 4 3 1
4 6 1 8 5 3 7 2 9
```

Podemos verificar pela imagem que se segue que a solução está de facto correta.

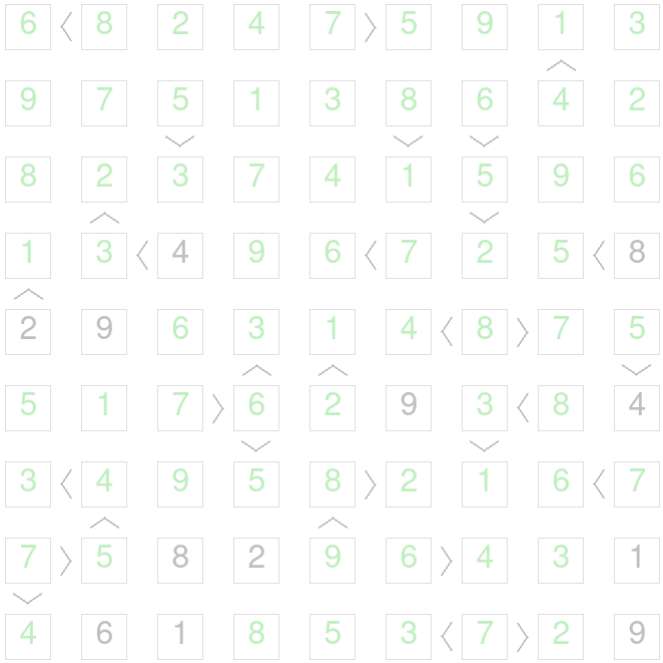

Figura 3: Puzzle futoshiki 9x9 resolvido.