

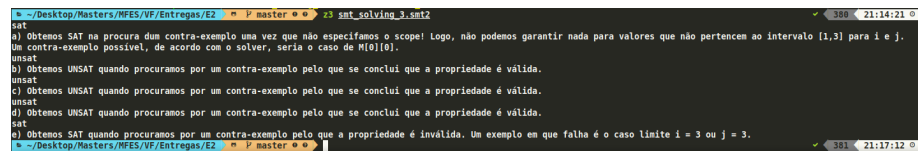
# SMT Solving

Pedro Costa A85700

March 2021

## 1) Matriz

Os resultados para a validade das propriedades está incluído no script `smt2`. No entanto, para facilitar a visualização, incluo aqui um print do resultado que se obtém ao correr a alínea 3:



```
~/Desktop/Masters/MES/VP/Entregas/E2  P master  3 smt2_smt2
sat
a) Obtemos SAT na procura dum contra-exemplo uma vez que não especificamos o scope! Logo, não podemos garantir nada para valores que não pertencem ao intervalo [1,3] para i e j.
Um contra-exemplo possível, de acordo com o solver, seria o caso de M[0][0].
unsat
b) Obtemos UNSAT quando procuramos por um contra-exemplo pelo que se conclui que a propriedade é válida.
unsat
c) Obtemos UNSAT quando procuramos por um contra-exemplo pelo que se conclui que a propriedade é válida.
unsat
d) Obtemos UNSAT quando procuramos por um contra-exemplo pelo que se conclui que a propriedade é válida.
sat
e) Obtemos SAT quando procuramos por um contra-exemplo pelo que a propriedade é inválida. Um exemplo em que falha é o caso limite i = 3 ou j = 3.
```

Figure 1: Resultados obtidos relativos à validade das propriedades

## 2) Puzzle Solver - Survo

O puzzle que escolhi resolver foi o Survo. A resolução deste passa, de forma simples, por resolver um sistema de equações. SMT pareceu-me capaz de lidar com isso bastante bem, o que motivou a minha escolha. Escolhi, mais uma vez, utilizar Haskell para resolver o problema proposto, utilizando a biblioteca [Z3](#) indicada.

Apresento, de seguida, quais as restrições que considerei e como resolvi cada uma delas no programa.

### Variáveis

Para representar as variáveis recorri à representação em string separada por um underscore de cada par linha coluna. Ou seja, a posição (2,3) é representada pela string "2\_3". Guarda estas variáveis num dicionário para facilitar o acesso às mesmas.

```
vars <- mapM mkFreshIntVar $ p2s <$> v
let dict = M.fromList $ zip v vars
```

Figure 2: Representação das variáveis

### Os valores pertencem ao intervalo legal

Uma das condições impostas pelo puzzle é que qualquer valor deve pertencer ao intervalo compreendido entre 1 até  $colunas * linhas$ .

Em Haskell, podemos exprimir isto da seguinte forma, utilizando a função *limit* que bloqueia um valor entre um limite superior e um limite inferior:

```
assert <<= mkAnd <<= sequence (limit (head ints) (last ints) <<= vars)
```

Figure 3: Valores limitados

### Todos os valores têm de ser distintos

Em Haskell, isto foi expresso muito facilmente na linha apresentada a seguir:

```
assert <<= mkDistinct vars
```

Figure 4: Valores distintos

## Alguns valores estão inicialmente colocados

Para definir esta restrição apenas temos de dizer que a célula em que o valor está equivale ao valor em causa.

A seguinte linha de Haskell resolve este problema:

```
assert =<< mkAnd =<< mapM (uncurry mkEq . ((dict M.!) *** ((ints!!) . pred))) locked
```

Figure 5: Valores iniciais

## Objetivos de linha e coluna

A parte crucial dos puzzles Survo é que os valores colocados numa linha/coluna devem somar a um certo valor, um objetivo.

Trivialmente, basta dizer que todos os elementos que compõem a linha ou coluna em causa, quando somados, são equivalente ao tal objetivo.

Em Haskell isto obrigou a manipular acessos de forma coordenada pelo que utilizei duas funções auxiliares *access*, *setGoal*. A função *access*, como o nome tenta indicar, faz um aceso a uma linha ou uma coluna, retornando a lista de índices que compõem a mesma. A função *setGoal* tira partido da função *access*, estabelecendo então a equivalência necessária.

```
-- Right retrieves a line, Left retrieves a column
access (l,c) (Right x) xs = drop ((x-1)*c) . take (x*c) $ xs
access (l,c) (Left x) xs = map ((xs!!) . pred) (enumFromThenTo x (x+c) (l*c))

setGoal dim v lc g = assert =<< join (liftM2 mkEq (mkInteger g) (mkAdd $ access dim lc v))
```

Figure 6: access e setGoal

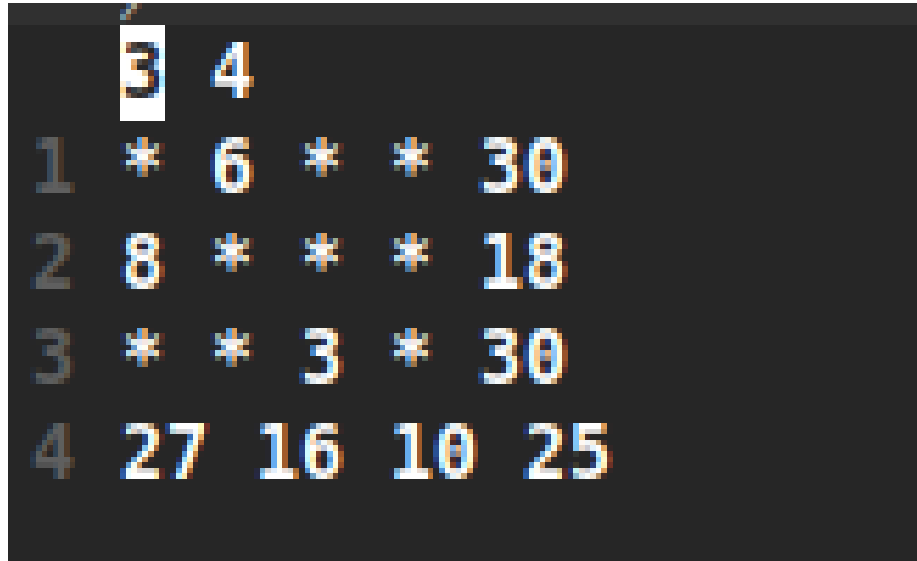
São utilizadas posteriormente sob uma lista de objetivos, definindo assim a última condição necessária.

```
mapM (uncurry (setGoal dim vars)) goals
```

Figure 7: Estabelecimento de objetivos

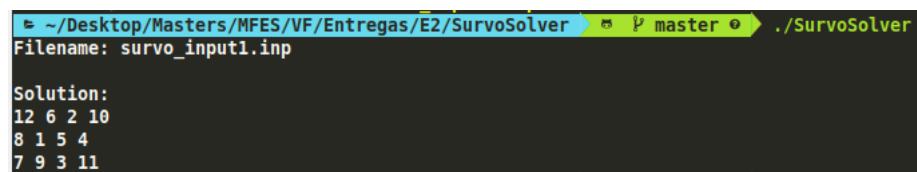
## Funcionalidades

Como era suposto, desenvolvi a capacidade de ler de um ficheiro um puzzle, utilizando o formato apresentado abaixo, transformando-o depois numa estrutura prática para o cálculo das restrições. A solução é impressa para o terminal, como se pode ver no exemplo de baixo.



```
3 4
1 * 6 * * 30
2 8 * * * 18
3 * * 3 * 30
4 27 16 10 25
```

Figure 8: Ficheiro de input



```
~/Desktop/Masters/MFES/VF/Entregas/E2/SurvoSolver master ➤ ./SurvoSolver
Filename: survo_input1.inp
Solution:
12 6 2 10
8 1 5 4
7 9 3 11
```

Figure 9: Execução do programa