



Universidade do Minho

MathemaGrids Solver implementation using Z3 SMT Solver

José Pereira pg27748
Marta Azevedo pg27763
Tiago Brito pg27724

25 de Abril de 2015

1 MathemaGrids

O MATHEMAGRIDS é um puzzle onde o objectivo é preencher uma tabela $m \times m$ com inteiros entre 1 e m , de tal maneira que cada um desses números aparece apenas uma vez.

Para além disso, a posição dos números deve respeitar as operações que já estão presentes no tabuleiro.

Neste relatório, pretendemos descrever de forma clara a nossa implementação.

Para isso vamos usar como exemplo o seguinte tabuleiro:

7	x		—		=	38
x		+		+		
	—		—		=	1
—		x		—		
	x		x		=	27
=		=		=		
55		72		6		

Neste exemplo, já é dada uma *hint*, o 7 que aparece na primeira linha, primeira coluna.

2 Implementação

2.1 SMT Solver

Como SMT-SOLVER estamos a usar o Z3 Theorem Prover da Microsoft Research. Este, é usado em vários softwares de verificação formal.

2.2 Linguagem de Programação

Como linguagem de interface com o Z3 preferimos usar o PYTHON Z3PY devido à sua API fácil de usar e devido a ser uma linguagem simples e eficaz para o que queríamos fazer.

2.3 Interface Gráfica

No que toca à Interface Gráfica, foi feito um trabalho de pesquisa prévio pelas mais diversas frameworks para python e, no final, decidimos usar o KIVY v1.9.0.

KIVY trata-se de uma Framework multi-plataforma para Python, correndo em Linux, Windows, OS X, Android e iOS. É uma ferramenta grátis e é "GPU Accelerated".

3 Estrutura

O Projeto é dividido em cinco módulos:

- `game.py`
Este é o módulo principal do projecto, sendo responsável pela inuficação de todas as funcionalidades e funcionamento das mesmas.
- `gameGUI.py`
Este é o módulo responsável por toda a interface gráfica. É ele quem inicializa o jogo e utiliza os restantes módulos de forma a executar as respetivas funções.
- `generate_board.py`
Trata-se do módulo responsável por criar tabuleiros e gerar as suas soluções utilizando o *Z3*.
- `parsing.py`
É o módulo responsável por fazer a leitura de um ficheiro de texto. Neste módulo está implementada uma gramática (que utiliza a biblioteca *pyparsing*) de forma a validar o input do ficheiro.
- `solveMathemaGrids.py`
Este trata-se de um módulo de suporte onde temos algumas funções auxiliares bem como a função que nos permite usar o *Solver* para obter a solução de um tabuleiro lido de ficheiro.

4 Execução

Para a execução do programa basta executar na consola :

```
$ python game.py
```

Existem flags opcionais que podem ser usadas:

- - **-console** : permite a execução do software em modo consola. Deve ser seguido do nome/localização do ficheiro de input

E tendo atenção às seguintes dependencias:

- **Python** : Disponível já de raiz com o Linux e OSX, mas também disponível para Windows.

- **Kivy** : Necessário *Kivy* v1.9.0 disponível no website <http://kivy.org/>.
Para instalar:

```
$ sudo add-apt-repository ppa:kivy-team/kivy
$ sudo apt-get install python-kivy
```

- **Z3** : Disponível em <https://github.com/Z3Prover/z3>:
- **pyparsing** : Disponível em <http://pyparsing.wikispaces.com/> e com instalação simples através do pip, bastando fazer

```
$ pip install pyparsing
```

5 Ficheiros input

Para representar um tabuleiro de MATHEMAGRIDS (o do exemplo) usamos o seguinte ficheiro de texto:

```
7*.-.=38
*,+,+
.-.-.=1
-,*,-
.*.*.=27
=,=,=,=
55,72,6
```

Os "." representam os espaços que têm que ser preenchidos com números e as "," representam os espaços que, apesar de existirem no tabuleiro, não podem ser inseridos com números.

Também estão representadas as operações sendo o * a multiplicação, o + a soma, - a subtração e / a divisão.

5.1 Codificação do MATHEMAGRIDS

Para codificar o puzzle, usamos as seguintes propriedades (descritas em <https://www.brainbashers.com/mathemagrids.asp>):

1. Usar todos os dígitos de 1 a $m * m$ (no nosso exemplo, até 9);
2. Nenhum número pode ser repetido;
3. As operações são feitas da esquerda para a direita e de cima para baixo, sendo a ordem de prioridade normal da matemática ignorada;
4. Não podem existir divisões por 1 nem multiplicações por 1;
5. Em nenhum ponto, os resultados intermédios do cálculo são valores inferiores a zero.

As variáveis *x_membros* e *y_membros* representam a quantidade de espaços em que o jogador pode inserir números nas linhas e nas colunas respectivamente. As regras usadas são:

1. `max_size = [And(1 <= x[j][i], x[j][i] <= size * size)
for i in range(size) for j in range(size)]`

2. (No ficheiro `generate_board.py`)

```
unique = [Distinct([x[j][i] for i in range(size) for j in range(size)])]
```

3. Para garantir que as operações são feitas na ordem correta, escrevemos as equações horizontais e verticais que, neste exemplo são :

- **Equações Verticais:**

```
(7 * x_1_2) - x_1_3 = 55  
(x_2_1 + x_2_2) * x_2_3 = 72  
(x_3_1 * x_3_2) - x_3_3 = 6
```

- **Equações Horizontais:**

```
(7 * x_2_1) - x_3_1 = 38  
(x_1_2 - x_2_2) - x_3_2 = 1  
(x_1_3 * x_2_3) * x_3_3 = 27
```

4.

```
not_one_exceptions = []  
for i in range(len(board)):  
    for j in range(len(board[i])):  
        if str(board[i][j]) == "/" or str(board[i][j]) == "*":  
            if i % 2 == 0:  
                not_one_exceptions.append(Not(x[i/2][(j+1)/2] == 1))  
            else:  
                not_one_exceptions.append(Not(x[(i+1)/2][j/2] == 1))
```

5. Como a operação crítica para isto acontecer é apenas a subtração:

```
bigger_than_zero = []  
for i in range(len(board)):  
    for j in range(len(board[i])):  
        if i % 2 == 0:  
            if board[i][j] == '-':  
                if j == 1:  
                    bigger_than_zero.append(Not(x[i/2][(j-1)/2] - x[i/2][(j+1)/2] < 0))  
            else:  
                equation = ""  
                for n in range(j):  
                    if n > 2 and str(board[i][n]) != "." and not(board[i][n].isdigit()):  
                        equation = "("+equation+")"+str(board[i][n])  
                else:  
                    if board[i][n] == "." or board[i][n].isdigit():
```

```

equation += "x["+str(i/2)+"]["+str(n/2)+"]"
else:
    equation += board[i][n]
equation = equation + "-" + "x[" + str(i/2) + "]" + str((j+1)/2)+"]" + ">0"
bigger_than_zero.append(eval(equation))

```

6. Esta regra serve para garantir que o y em x/y é um múltiplo do x . Caso contrário, por exemplo $6/2 = 3 = 7/2$

```

multiple = []
for i in range(len(board)):
    for j in range(len(board[i])):
        if i % 2 == 0:
            if board[i][j] == '/':
                multiple.append(x[i/2][(j-1)/2] % x[i/2][(j+1)/2] == 0)
        else:
            if board[i][j] == '/':
                multiple.append(x[(i-1)/2][j/2] % x[(i+1)/2][j/2] == 0)

```

No final, é enviado para o solver:

```
rules = unique+bigger_than_zero+not_one_exceptions+max_size+multiple
```

6 *generate_board.py*

6.1 Hints

As hints foram adicionadas ao nosso projeto com o objetivo de ajudar o jogador a chegar mais rapidamente à solução. Como tal, na opção “*Settings*” podemos seleccionar duas opções relativamente a este assunto:

1. Ativar hints
2. Hints preenchem apenas campos vazios

Se a **opção 1** estiver ativada, quando clicarmos em “*Hint*” no menu inicial é sugerido um número para um espaço (mesmo que este já tenha sido preenchido pelo utilizador).

Se ativarmos a **opção 2**, as sugestões são apenas dados em campos que não tenham sido preenchidos, mesmo que as opções que o utilizador inseriu estejam erradas.

6.2 Geração de tabuleiros

A geração de tabuleiros é feita recebendo como argumento o tamanho do tabuleiro e é criada uma matriz com esse tamanho. Por exemplo, para gerar um tabuleiro do mesmo tamanho que o exemplo, o número seria 3.

As operações estão guardadas num array e, de forma aleatoria, são colocadas na matriz.

A seguir, e em conjunto com as regras da secção **3.3** é enviada a matriz ao *solver*. Caso o problema seja satisfazível, é considerada essa matriz uma matriz válida. É enviada para uma função que calcula os valores que vão estar na última linha e última coluna (depois do =).

Depois do tabuleiro pronto, é enviado para um ficheiro de texto onde é armazenada com especificado na secção **3.1**.