A solver for Hadoku using Z3

Daniel Novais pg27745
Eduardo Pessoa pg25314

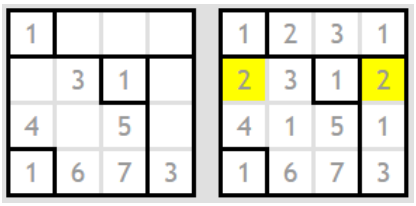
25 de Abril de 2015

1 Hadoku

O HADOKU é um puzzle, com certas semelhanças com o famoso SUDOKU onde o objetivo é preencher uma tabela *ixj* que está dividida em áreas irregulares. Cada área deve ser preenchida com inteiros entre 1 e o número de células presentes nessa área, sem repetições.

Mas uma pequena alteração torna o HADOKU num jogo completamente diferente: Um número N pode aparecer mais do que uma vez na mesma linha (ou coluna) se a distância entre as duas células for, pelo menos, N.

Por exemplo, entre dois números 1 na mesma coluna tem de haver pelo menos uma célula de distância, entre dois números 2 duas células e por aí em diante. Para melhor se compreender o jogo segue um exemplo de um puzzle no seu estado inicial e no seu estado final:



2 Z3

Como SMT-SOLVER decidimos utilizar o Z3 *Theorem Prover* da *Microsoft Research*. O Z3 é usado em várias plataformas de verificação formal.

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 nosso objetivo.

3 Implementação

O nosso ficheiro *py*, *hadoku.py* é invocado como um ficheiro *python* qualquer, por exemplo:

```
$ python hadoku.py h1.txt
```

É necessário que o z3 esteja instalado e conectado com o PYTHON.

3.1 Formato de input

Para representar um tabuleiro de HADOKU usamos, por exemplo, o seguinte ficheiro de texto:

```
areas
1 1 2 2
3 1 1 2
4 1 1 5
4 6 6 6
board
. . . .
1 3 5 .
. . 6 1
. . . .
END
```

No ficheiro de *input* estão representadas duas tabelas, que possuem a informação necessária para representar um tabuleiro de HADOKU. O ficheiro possui também *keywords* para ajudar a identificar essas tabelas.

A primeira tabela, representa as áreas do tabuleiro. Cada área é representada por um número (1, 2, 3, ..., 9). O número máximo de áreas possível é nove, já que não encontramos nenhum *puzzle* com mais do que nove áreas.

A segunda tabela (que começa com a *keyword* "*board*") representa as células do tabuleiro no estado inicial, sendo o valor das células com $'.'$ desconhecido.

3.2 Codificação do HADOKU.PY

Para codificar o puzzle, começamos por ler as duas tabelas do ficheiro de *input* para duas matrizes de *python*. Uma contêm a primeira tabela (áreas) e outra contem a segunda (células).
De seguida criamos uma lista com variáveis para representar cada posição do tabuleiro:

```
cells = [ [ Int(\"h_\%s_\%s" % (i+1, j+1)) for j in range(yy) ]
        for i in range(xx) ]
```

Podemos começar agora a introduzir *asserts* no nosso *Solver*. Os primeiros (e mais simples) são inserir os valores das células que já conhecemos:

```
for y in range(0,xx):
    for x in range(0,yy):
        ...
        if vals[y][x] != '.':
            ...
            s.add(cells[y][x] == vals[y][x])
```

De seguida adicionamos as restrições de cada célula relativas à sua área, isto é, cada célula tem um valor maior que 0 e menor ou igual ao tamanho da área onde se encontra:

```
for x in range(xx):
    for y in range(yy):
        s.add(And(1 <= cells[x][y],
        cells[x][y] <= maxArea(Board[x][y])))
```

O número de células de uma área é obtido com a função *maxArea*(). Adicionamos também ao Solver as restrições que dizem que em cada área, todas as células têm valores diferentes:

```
for x in range(xx):
    for y in range(yy):
        s.add(Distinct(getArea(x,y)))
```

A função *getArea*() devolve uma lista de todos os elementos da área de uma célula.
Por fim temos de adicionar as restrições de dois números iguais poderem aparecer aparecer na mesma linha/coluna em situações específicas (como referido em cima):

```
for x in range(xx):
    for y in range(yy):
        if vals[x][y] != '.':
            row, v = getRows(vals[x][y], x, y)
            for l in range (len(row)):
                s.add(row[l] != v)
            col, v = getCols(vals[x][y], x, y)
            for l in range (len(col)):
                s.add(col[l] != v)
        else :
            for i in range (1, 1 + maxArea(Board[x][y])):
                row, v = getRows(i, x, y)
                for l in range (len(row)):
                    s.add(Implies(cells[x][y] == i, row[l] != v))
                col, v = getCols(i, x, y)
                for l in range (len(col)):
                    s.add(Implies(cells[x][y] == i, col[l] != v))
```

Este pedaço de código itera por todas as células do tabuleiro, e:

- Se a célula tiver valor conhecido (!= '.'), digamos N, então para todas as células que se encontrem na mesma linha e coluna dessa célula, introduzimos asserções no *Solver* que impedem que as N primeiras células para cada direção (cima, baixo, esquerda, direita), possuam o valor N. No exemplo descrito no início deste relatório, como a primeira célula possui o valor 1 então teríamos duas asserções que diriam:

  1. h_1_1 != h_2_1
  2. h_1_1 != h_1_2

- Caso contrário (== '.') decidimos optar por criar um conjunto de asserções $if - then$ que cobrem todas as possibilidades de valores de uma célula com valor desconhecido. Para isso utilizamos a função do z3 *Implies*. Para todos os valores que uma célula pode obter, considerando as restrições da área, criamos restrições $if - then$ que façam com que os valores obedeçam ao caso referido no ponto acima. Por exemplo, no tabuleiro referido no primeiro capítulo, para a célula desconhecida h_2_4 (2^a linha, 4^a coluna) vamos ter:

  1. Implies(h_2_4 == 1, h_3_4 != 1)
  2. Implies(h_2_4 == 1, h_1_4 != 1)
  3. Implies(h_2_4 == 2, h_3_4 != 2)
  4. Implies(h_2_4 == 2, h_1_4 != 2)
  5. Implies(h_2_4 == 3, h_3_4 != 3)
  6. Implies(h_2_4 == 3, h_1_4 != 3)
  7. Implies(h_2_4 == 3, h_2_1 != 3)

  Que cobre todos os possiveis casos.

Cabe agora ao *Solver* analisar todas as asserções passadas e tentar encontrar um modelo que as satisfaça todas. Se tal modelo não for encontrado podemos concluir que o problema é insatisfazível:

```
if checkSat(s):
    m = s.model()
    print "Modelo encontrado!"
else:
    print "Impossivel de resolver"
```

4 Pontos Extra

Como ponto extra decidimos implementar a possibilidade de um utilizador jogar uma partida de HADOKU. No modo de jogo, a cada jogada, o utilizador é presenteado com duas tabelas: uma para os números não conhecidos e outra para as áreas (tal como no ficheiro de input):

```
    A B C D

1 | 1 . . .

2 | . 3 1 .

3 | 4 . 5 .

4 | 1 6 7 3

-----------------
    A B C D

1 | 1 2 2 2

2 | 3 3 4 5

3 | 3 3 3 5

4 | 6 3 3 5

Próximo movimento [p.ex. B3-2] (s=sair) (d=desistir):
```

De seguida é pedido ao utilizador para introduzir uma célula e um valor para essa célula, por exemplo, $B3 - 2$ onde $B3$ é a célula e 2 é o valor a inserir. Depois disto o solver vai ver se é possível encontrar um modelo válido com a jogada do utilizador. Se for possível aceita a jogada, se não for, avisa que a jogada é inválida. O jogo continua da mesma forma até o utilizador chegar ao fim. Também é possível desistir a meio e ver logo a solução final.

```
    A B C D

1 | 1 2 3 1

2 | 2 3 1 2

3 | 4 1 5 1

4 | 1 6 7 3
```