

UNIVERSIDADE FEDERAL RURAL DO RIO
DE JANEIRO

CIÊNCIA DA COMPUTAÇÃO

ATIVIDADE ACADÊMICA DE INTELIGÊNCIA ARTIFICIAL

Resolvendo Sudoku 9x9 com o algoritmo A*

Autor:

Lívia de Azevedo da
Silva

Orientador:

Dr. Marcelo Panaro de
Moraes Zamith

Julho de 2015

Sumário

1	Introdução	4
2	Motivação da escolha do tema	5
3	Dificuldades na implementação	5
4	Desenvolvimento do projeto	5
5	Descrição detalhada da técnica utilizada	7
5.1	O elemento $G(x)$	7
5.2	O elemento $H(x)$	8
5.3	O elemento $F(x)$	9
5.4	Lista aberta e lista fechada	9
5.5	O algoritmo A^* para Sudoku: a função recursiva <i>resolver()</i> . . .	9
6	Testes de validação	13
6.1	Casos que resolvam Sudoku's com solução	14
6.2	Caso que tenta resolver Sudoku sem solução	17

Lista de Figuras

1	Exemplo de um Sudoku 9x9	4
2	O slot com contorno azul é o slot da lista aberta analisado e os slots com contornos vermelhos representam os 20 slots somando linha, coluna e subgrade. A função $g(x)$ é definida como: $g(x) = 20 - \text{slotsAbertosLinhaColunaSubgrade}$	8
3	Primeiro exemplo com solução	14
4	Segundo exemplo com solução	15
5	Terceiro exemplo com solução	16
6	Exemplo Sudoku sem solução	17

Listings

1	Código do método <i>resolver()</i>	9
---	--	---

1 Introdução

Primeiramente, introduzirá as regras e conceitos do jogo Sudoku:

“Sudoku é um quebra-cabeça baseado na colocação lógica de números. O objetivo do jogo é a colocação de números de 1 a 9 em cada uma das células vazias numa grade de 9x9, constituída por 3x3 subgrades chamadas regiões. O quebra-cabeça contém algumas pistas iniciais, que são números inseridos em algumas células, de maneira a permitir uma indução ou dedução dos números em células que estejam vazias. Cada coluna, linha e região só pode ter um número de cada um dos 1 a 9. Resolver o problema requer apenas raciocínio lógico e algum tempo.”

Wikipédia.

(<http://pt.wikipedia.org/wiki/Sudoku>)

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figura 1: Exemplo de um Sudoku 9x9

Vale lembrar que o Sudoku de ordem 9 (9x9) é o que possui mais de jogos construídos prontos para serem resolvidos (bilhões de possibilidades a disposição!) e o mais comumente jogado. Além do ordem 9, existem o de ordem 3 e o de ordem 4, por exemplo.

O Sudoku será o objeto de estudo neste projeto. Aplicaremos o método de busca A* para resolver qualquer tipo de situação de jogo do Sudoku 9x9. Os detalhes do projeto e como foi implementado serão explicados mais adiante.

2 Motivação da escolha do tema

O Sudoku é um jogo versátil no quesito de que qualquer pessoa tenha a facilidade de jogá-lo e entender suas regras, um dos motivos que um tema como esse possa ser melhor compreendido quanto a ideia aplicada de Inteligência Artificial para resolvê-lo.

Outra questão se refere a popularidade do Sudoku. Muitas pessoas, com certeza, já jogaram ou, pelo menos, já ouviram falar sobre o mesmo. Além disso, o Sudoku é um jogo que requer uma boa observação de jogo e certo raciocínio lógico, ou seja, um bom e interessante problema e exemplo para um computador resolver com uma técnica de Inteligência Artificial.

3 Dificuldades na implementação

O algoritmo A* é normalmente aplicado em problemas que exigem o cálculo do menor caminho de um ponto em relação a outro. Portanto, remodelar a ideia para o ambiente do Sudoku foi o desafio de implementação do trabalho.

4 Desenvolvimento do projeto

O projeto foi implementado na linguagem de programação Java, com suporte a programação Orientada a Objetos. O trabalho foi feito na IDE (Ambiente de Desenvolvimento Integrado) Eclipse. O projeto contém quatro arquivos, os quais são:

1. **Main.java:** Contém a classe Main que contém o método principal *main* onde será executado o programa. É aqui que será encontrado as variáveis (tabuleiros) de testes para a validação do programa;

2. **Slot.java:** Contém a classe Slot. O slot representa cada subgrade (quadrado pequeno) no tabuleiro do Sudoku, o qual será nosso principal objeto de manipulação do nosso algoritmo A*.

A classe Slot terá os seguintes atributos:

- Uma booleana chamada *preenchido*, informando se o slot está vazio ou não;
- Um inteiro chamado *fn*, armazenando o valor do resultado da heurística do A*;
- Dois inteiros chamados *i* e *j*, representando a posição da linha e coluna, respectivamente;
- Um vetor de inteiros chamado *possibilidades*, indicando os valores válidos naquele slot.

Além dos atributos, teremos os métodos da classe que são os *getters* (que retornam os valores do determinado atributo) e *setters* (que modificam os valores de determinado atributo) referentes a cada atributo.

3. **Funcoes.java:** Contém a classe Funcoes. Esta classe agrupará todos os métodos que serão necessários para o funcionamento do algoritmo.

Os métodos inclusos são:

- (a) **Gn():** Calcula o valor de $g(n)$;
- (b) **Hn():** Calcula o valor de $h(n)$;
- (c) **imprimirResultado():** Imprimir o tabuleiro do Sudoku que foi mandado como parâmetro da função;
- (d) **bubbleSort():** Função referente ao método de ordenação *bubble sort*. O *bubble sort* foi implementado de modo a ordenar os valores de forma decrescente e foi adaptado para ordenar um vetor de objetos Slots de acordo com o valor do $f(n)$ de cada um;
- (e) **preencheuTudo():** Verifica se o tabuleiro foi preenchido por completo. Se isto ocorrer, temos uma solução válida para o Sudoku analisado.

4. **Resolver.java:** Contem a classe Resolver que tem o método principal do projeto: `resolver()`. Este método será o que executará o algoritmo A* de resolução do Sudoku.

5 Descrição detalhada da técnica utilizada

O A^* é um método de busca heurística que tem como base de medida do melhor caminho (opção) a fórmula $f(x) = h(x) + g(x)$, sendo $h(x)$ uma função heurística que determina a “distância” da posição atual até o objetivo e $g(x)$ uma função que calcula um “peso” para que determinado movimento ocorra. Além destes parâmetros, tem-se a lista aberta, que contém todos os caminhos que se podem percorrer, e a lista fechada, que contém os passos já tomados pelo algoritmo a qual no final retornará o melhor caminho. No Sudoku não se tem essa ideia de se descobrir uma menor distância em metros, centímetros, milímetros etc. para se determinar uma melhor escolha. Mas há a possibilidade de abstrair esta ideia e levá-la ao contexto do Sudoku. Assim, $g(x)$, $h(x)$ e $f(x)$ poderiam ser determinados e uma medida heurística seria determinada.

5.1 O elemento $G(x)$

O cálculo de $g(x)$ levou em consideração a subtração do o número total de slots da linha, coluna e subgrade onde o slot analisado está, que será sempre 20 (linha = 8; coluna = 8; subgrade = 4) com o número de slots disponíveis naquele momento, aplicando a mesma ideia de verificação. A ideia considera que quanto mais slots abertos existir na linha, coluna e subgrade menor será a probabilidade de preencher corretamente o slot vazio, sendo que o contrário também é válido. Isto se deve pois necessita-se saber os números preenchidos dos outros slots analisados na linha, coluna e subgrade para se obter uma maior confiabilidade de inserção de um certo número no slot.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figura 2: O slot com contorno azul é o slot da lista aberta analisado e os slots com contornos vermelhos representam os 20 slots somando linha, coluna e subgrade. A função $g(x)$ é definida como: $g(x) = 20 - \text{slotsAbertosLinha-ColunaSubgrade}$.

5.2 O elemento $H(x)$

O cálculo de $h(x)$ analisa a quantidade de números possíveis que se pode inserir no slot, ou seja, o valor de $h(x)$ varia entre 0 e 9 (pois temos as possibilidades: 1,2,3,4,5,6,7,8 ou 9). A ideia é ter-se uma noção de quantas possíveis inserções se tem e assim obter mais um critério de avaliação do quão bom seria a escolha do slot aberto naquele momento. Quanto menor o valor de $h(x)$, mais perto da solução correta se está.

5.3 O elemento $F(x)$

O cálculo de $f(x)$ se dá pela soma do $g(x)$ com o $h(x)$, unindo os critérios utilizados para cada função e com isto obtendo um valor determinístico que informará a qualidade de se preencher aquele slot. O slot escolhido será aquele que possuir o maior valor de $f(x)$.

5.4 Lista aberta e lista fechada

A lista aberta corresponde aos slots abertos que existem no tabuleiro e a lista fechada os slots já preenchidos.

5.5 O algoritmo A^* para Sudoku: a função recursiva *resolver()*

A função recursiva *resolver()* é a que contém toda a ideia do A^* que foram explicadas. Ela receberá três parâmetros: o tabuleiro a ser resolvido, a lista aberta e o número de slots abertos que existem (quantidade de elementos na lista aberta). Com essas informações ela executa o A^* . A função calculará os valores de $f(x)$ dos slots abertos disponíveis e ordenará a lista aberta de forma decrescente e escolherá o primeiro elemento da lista aberta ordenada (o que possuir o maior $f(x)$) o qual será o melhor slot a ser preenchido no estado correspondente do tabuleiro. A partir da escolha do melhor slot, preenche-se o slot vazio com o primeiro número do vetor de possibilidades do slot que é válido e então colocando este slot na lista fechada. Devido a implementação da recursividade, a função será usada novamente recebendo os parâmetros do novo tabuleiro com o slot preenchido e a lista aberta modificada e irá se repetir o mesmo processo. Se esse preenchimento for incorreto, em algum momento mais a frente haverá uma incompatibilidade de preenchimento do tabuleiro e teremos um retorno *false* da função; caso realmente exista uma resposta, este ciclo acontecerá até que todos os casos sejam válidos. Quando o último slot vazio do tabuleiro for preenchido, que será determinado pela função *preencheuTudo()* no início da função, retornará-se o valor booleano *true* para determinar o fim do algoritmo e a obtenção de um resultado. Caso não haja uma solução para o Sudoku, pelo menos um slot do tabuleiro não terá uma resposta que satisfaça o jogo e assim obrigará a função a retornar sempre *false* quando se analisar o estado deste slot: portanto o retorno final da função será *false*.

Tendo uma resposta final, o programa imprimirá o resultado encontrado.

Listing 1: Código do método *resolver()*

```

1 public class Resolver {
3     public static final int DIMENSAO = 9;
4     public static final int
5         NUMERO_TOTAL_SLOTS_LINHA_COLUNA_QUADRADO =
6         20;
7     public static final int DIMENSAO_QUADRADOS = 3;
8     public static final int LISTA_FECHADA = -1; //
9         Representa o elemento da lista aberta que foi
10        preenchido.
11
12    public static boolean resolver(int[][] tabuleiro
13        ,Slot[] listaAberta, int numeroSlotsAbertos){
14
15        int i,j;
16        Slot[] listaAbertaDaIteracao = new Slot[
17            numeroSlotsAbertos];
18        int[][] tabuleiroIteracao = new int[DIMENSAO
19            ][DIMENSAO];
20
21        //Fazendo uma copia do tabuleiro original da
22        iteracao
23        for(i = 0;i < DIMENSAO;i++){
24            for(j = 0;j < DIMENSAO;j++){
25                tabuleiroIteracao[i][j] = tabuleiro[
26                    i][j];
27            }
28        }
29        //Se o tabuleiro foi todo preenchido
30        seguindo as regras, imprimi-se o
31        resultado.
32        if(Funcoes.preencheuTudo(tabuleiro)){
33            Funcoes.imprimirResultado(tabuleiro);
34            return true;
35        }
36    }
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

29 //Criando os slots abertos(os espacos vazios
    da tabuleiro) da iteracao.
31 for(i = 0;i < numeroSlotsAbertos;i++){
    listaAbertaDaIteracao[i] = new Slot(
        listaAberta[i].getPreenchido(),
        listaAberta[i].getI(),listaAberta[i].
        getJ());

33     for(j = 0;j < DIMENSAO;j++){
        listaAbertaDaIteracao[i].
            setPossibilidades(j,listaAberta[i]
                .getPossibilidades(j));

35     }
}
37 //Calculando o f(n).
for(i = 0;i < numeroSlotsAbertos;i++){
39     if(!listaAbertaDaIteracao[i].
        getPreenchido()){

41         for(j = 0;j < DIMENSAO;j++){
            listaAbertaDaIteracao[i].
                setPossibilidades(j,Funcoes.
                    ehPossivelInserir(
                        tabuleiroIteracao,j + 1,
                        listaAbertaDaIteracao[i]));

43         }

45         if(Funcoes.Hn(listaAbertaDaIteracao[
            i]) != 0){
            listaAbertaDaIteracao[i].setFn((
                Funcoes.Gn(
                    listaAbertaDaIteracao[i],
                    tabuleiroIteracao) + Funcoes.
                    Hn(listaAbertaDaIteracao[i]))
                );

47         }else{
            return false;

49         }
    }
51 }

```

```

53 //O Fn a ser escolhido ser o maior dentre
    todos!(O bubbleSort aplicado sera em
        ordem decrescente!)
    Funcoes.bubbleSort(listaAbertaDaIteracao,
        numeroSlotsAbertos);

55
57 for(j = 0; j < DIMENSAO; j++){
    if(listaAbertaDaIteracao[0].
        getPossibilidades(j) != 0){
        tabuleiroIteracao[
            listaAbertaDaIteracao[0].getI()][
            listaAbertaDaIteracao[0].getJ()]
            = listaAbertaDaIteracao[0].
                getPossibilidades(j);
59 listaAbertaDaIteracao[0].
        setPreenchido(true); //Representa
            o slot preenchido.
        listaAbertaDaIteracao[0].setFn(
            LISTA_FECHADA); //O slot vai para
            a lista fechada(-1 no fn
            representa o slot que esta na
            lista fechada).

61
        //Se conseguirmos resolver o
            tabuleiro, retornamos como
            verdadeiro nossa tentativa.
63 if(Resolver.resolver(
            tabuleiroIteracao,
            listaAbertaDaIteracao,
            numeroSlotsAbertos)){
            return true;
65 }

67 //Se nao entrarmos no if, entao o
            valor inserido no slot
            correspondente no eh valido.
            Precisamos trocar este valor.
        tabuleiroIteracao[
            listaAbertaDaIteracao[0].getI()][
            listaAbertaDaIteracao[0].getJ()]
            = 0;

```

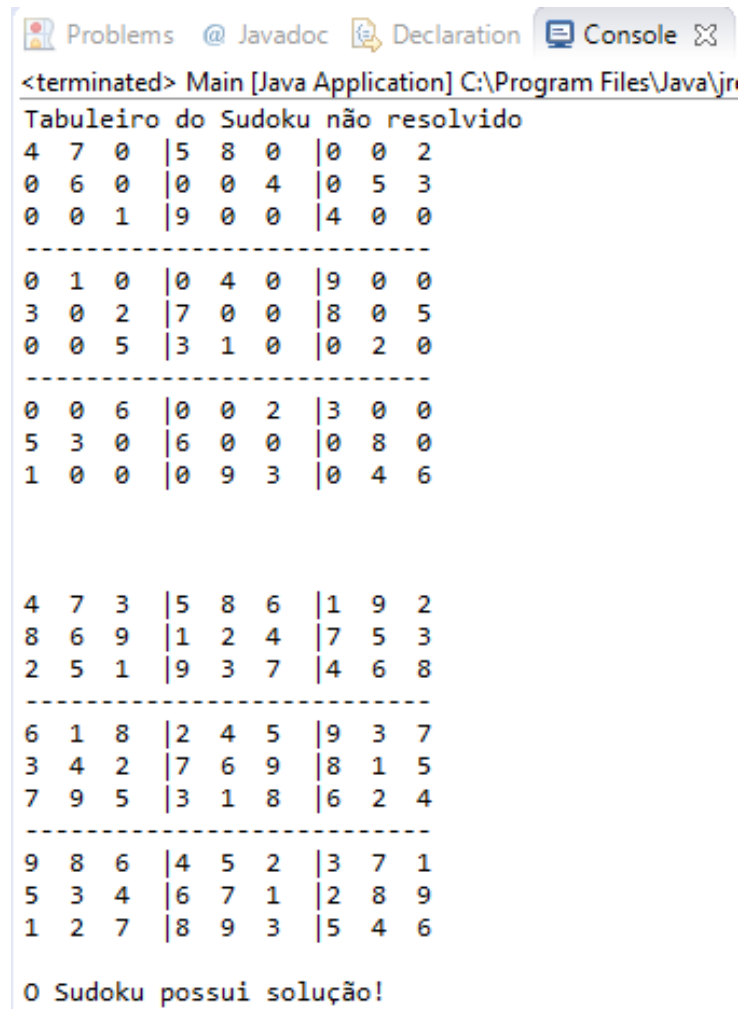
```

69         listaAbertaDaIteracao[0].
           setPreenchido(false);
71     }
    }
    //Se nenhum valor eh valido,entao temos um
    slot anterior preenchido de maneira
    errada.Precisamos voltar e mudar este
    valor.
73     return false;
    }
75 }
```

6 Testes de validação

O projeto possui quatro tabuleiros exemplos como casos de testes: três sendo tabuleiros válidos e um como tabuleiro que não possui solução. A seguir, tem-se os casos de testes usados para validar a funcionalidade do A* aplicado ao Sudoku 9x9.

6.1 Casos que resolvam Sudoku's com solução



```
<terminated> Main [Java Application] C:\Program Files\Java\jre
Tabuleiro do Sudoku não resolvido
4 7 0 | 5 8 0 | 0 0 2
0 6 0 | 0 0 4 | 0 5 3
0 0 1 | 9 0 0 | 4 0 0
-----
0 1 0 | 0 4 0 | 9 0 0
3 0 2 | 7 0 0 | 8 0 5
0 0 5 | 3 1 0 | 0 2 0
-----
0 0 6 | 0 0 2 | 3 0 0
5 3 0 | 6 0 0 | 0 8 0
1 0 0 | 0 9 3 | 0 4 6

4 7 3 | 5 8 6 | 1 9 2
8 6 9 | 1 2 4 | 7 5 3
2 5 1 | 9 3 7 | 4 6 8
-----
6 1 8 | 2 4 5 | 9 3 7
3 4 2 | 7 6 9 | 8 1 5
7 9 5 | 3 1 8 | 6 2 4
-----
9 8 6 | 4 5 2 | 3 7 1
5 3 4 | 6 7 1 | 2 8 9
1 2 7 | 8 9 3 | 5 4 6

O Sudoku possui solução!
```

Figura 3: Primeiro exemplo com solução

```

Problems @ Javadoc Declaration Console
<terminated> Main [Java Application] C:\Program Files\Java\jre
Tabuleiro do Sudoku não resolvido
3 0 0 | 0 0 0 | 0 1 2
0 8 0 | 0 1 0 | 0 0 6
0 0 6 | 0 0 7 | 0 0 0
-----
9 0 0 | 6 0 0 | 2 0 0
0 1 0 | 0 5 0 | 0 9 0
0 0 5 | 0 0 4 | 0 0 8
-----
0 0 0 | 5 0 0 | 7 0 0
8 0 0 | 0 3 0 | 0 2 0
6 3 0 | 0 0 0 | 0 0 9

3 7 4 | 8 6 5 | 9 1 2
5 8 2 | 4 1 9 | 3 7 6
1 9 6 | 3 2 7 | 8 4 5
-----
9 4 3 | 6 8 1 | 2 5 7
7 1 8 | 2 5 3 | 6 9 4
2 6 5 | 9 7 4 | 1 3 8
-----
4 2 1 | 5 9 8 | 7 6 3
8 5 9 | 7 3 6 | 4 2 1
6 3 7 | 1 4 2 | 5 8 9

O Sudoku possui solução!

```

Figura 4: Segundo exemplo com solução


```

Problems @ Javadoc Declaration Console
<terminated> Main [Java Application] C:\Program Files\Java\jre
|Tabuleiro do Sudoku não resolvido
2 0 0 |0 0 0 |0 0 6
0 4 0 |0 0 0 |0 5 0
0 0 5 |0 0 0 |8 0 0
-----
0 0 0 |9 0 7 |0 0 0
0 0 0 |0 1 0 |0 0 0
0 0 0 |6 0 2 |0 0 0
-----
0 0 9 |0 0 0 |6 0 0
0 5 0 |0 0 0 |0 3 0
8 0 0 |0 0 0 |0 0 2

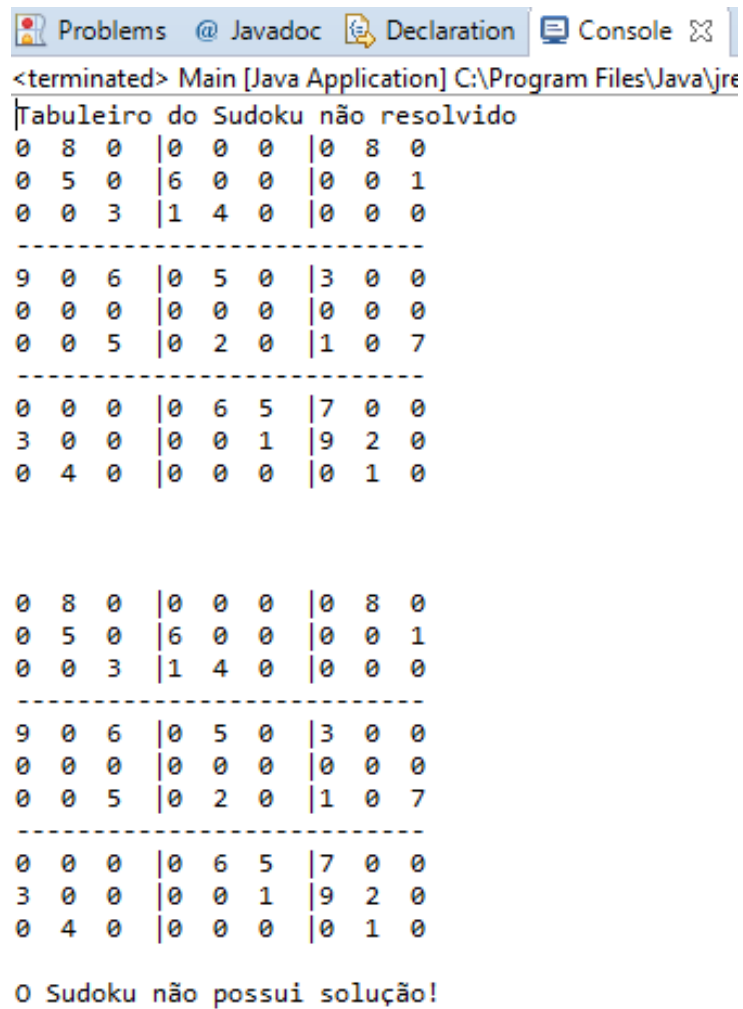
2 1 8 |3 5 9 |7 4 6
9 4 7 |8 6 1 |2 5 3
6 3 5 |7 2 4 |8 1 9
-----
1 8 2 |9 3 7 |4 6 5
4 9 6 |5 1 8 |3 2 7
5 7 3 |6 4 2 |9 8 1
-----
3 2 9 |1 8 5 |6 7 4
7 5 4 |2 9 6 |1 3 8
8 6 1 |4 7 3 |5 9 2

O Sudoku possui solução!

```

Figura 5: Terceiro exemplo com solução

6.2 Caso que tenta resolver Sudoku sem solução



```
Problems @ Javadoc Declaration Console
<terminated> Main [Java Application] C:\Program Files\Java\jre
|Tabuleiro do Sudoku não resolvido
0 8 0 |0 0 0 |0 8 0
0 5 0 |6 0 0 |0 0 1
0 0 3 |1 4 0 |0 0 0
-----
9 0 6 |0 5 0 |3 0 0
0 0 0 |0 0 0 |0 0 0
0 0 5 |0 2 0 |1 0 7
-----
0 0 0 |0 6 5 |7 0 0
3 0 0 |0 0 1 |9 2 0
0 4 0 |0 0 0 |0 1 0

0 8 0 |0 0 0 |0 8 0
0 5 0 |6 0 0 |0 0 1
0 0 3 |1 4 0 |0 0 0
-----
9 0 6 |0 5 0 |3 0 0
0 0 0 |0 0 0 |0 0 0
0 0 5 |0 2 0 |1 0 7
-----
0 0 0 |0 6 5 |7 0 0
3 0 0 |0 0 1 |9 2 0
0 4 0 |0 0 0 |0 1 0

O Sudoku não possui solução!
```

Figura 6: Exemplo Sudoku sem solução