Volume 5, número 1, dezembro de 2021
REVISTA DE TECNOLOGIA INVEST

Artigo 9

Métodos de Busca para Resolução de um Jogo de Sudoku

Ed Wilson Rodrigues Silva Júnior[1]
Higor Diniz Bravo[2]

RESUMO

Este artigo tem como intuito descrever uma implementação em Java para resolução de um jogo de Sudoku através de busca cega e heurística. Desse modo, partindo do uso de conceitos conhecidos sobre os assuntos apresentados e das especificações do trabalho, a solução se restringe a jogos de Sudoku 9x9 convencionais. Ademais, evidenciou-se que podemos usar diferentes técnicas para encontrar soluções deste jogo, podendo ser utilizada finitas plataformas e linguagens de programação para poder efetivar a sua implementação, não sendo restrito a apenas um método.

Palavras-Chave: Sudoku, Java, Métodos.

ABSTRACT

This article aims to describe a Java implementation to solve a Sudoku game through blind and heuristic search. Thus, starting from the use of known concepts about the subjects presented and the specifications of the work, the solution is restricted to conventional 9x9 Sudoku games. In addition, it was evidenced that we can use different techniques to find solutions of this game, and can be used finite platforms and programming languages to be able to implement them, not being restricted to only one method.

Keywords: Sudoku, Java, Methods.

[1] Doutorando em Computação Aplicada pela Universidade do Vale do Rio dos Sinos (Unisinos); Mestre em ensino de linguagens e seus códigos pelo Programa de Pós-Graduação Stricto Sensu em associação ampla entre a Universidade de Cuiabá-UNIC e o Instituto Federal de Educação, Ciência e Tecnologia do Estado de Mato Grosso-IFMT. Possui graduação em Sistemas de Informação pelo Centro Universitário de Várzea Grande, licenciatura em computação pelo Claretiano Centro Universitário e especialização em tecnologias na educação pela Universidade do Oeste Paulista. Tem experiência na área de ciência da computação, com ênfase em sistemas de computação, na educação profissionalizante e superior voltada para a área de tecnologia da informação e pesquisas em inovação, criatividade e metodologias de aprendizagem.
[2] Graduando em Análise e Desenvolvimento de Sistemas pela Faculdade Invest de Ciências e Tecnologia.

INTRODUÇÃO

O Sudoku (que significa “número único” em japonês) é um passatempo individual do tipo “quebra-cabeça” (puzzle), que consiste em uma grade de 9×9 células previamente preenchidas com alguns dígitos. O objetivo do jogo é preencher toda a grade de maneira que cada coluna, cada linha e cada uma das nove regiões de 3×3 casas contenham os dígitos de 1 a 9, uma e apenas uma vez cada. Além disso, esse *puzzle* pode ser aplicado com a finalidade de desenvolver e aumentar a capacidade de raciocínio lógico de seus usuários.

Este trabalho está organizado da seguinte forma: Na Seção 2 estão descritas as regras e técnicas de solução manual do problema que foram consideradas na implementação do trabalho. As decisões de implementação e abordagens estão descritas na Seção 3. Nas próximas seções são escalados os métodos utilizados para a implementação e desenvolvimento do mesmo.

Técnicas de Solução Manual

A inteligência artificial pode ser implementada a partir de uma gama de linguagens de programação, aos quais possuem determinadas especificidades para sua utilização, neste caso, optou-se por utilizar a linguagem Java devido a sua diversidade de pareamento e, segundo o *ranking* da RedMonk (RENATO, 2020), atualmente a linguagem de programação Java é a 3° linguagem mais popular do planeta, ficando atrás somente de C e *Python*. Através disso, o Java possui o JVM (*Java Virtual Machine*) sendo um elemento ao qual permite que essa linguagem possa ser rodada em diversas máquinas, a fim de expandir seu uso.

Portanto, este artigo irá focar no Sudoku, sendo um jogo que consiste em uma grade 9x9 separada em 9 blocos 3x3. Seu estado inicial é qualquer grade com espaços em branco que ainda mantenha um estado válido. Um estado válido é aquele onde não há repetição de número em uma ou mais colunas, linhas ou blocos. Dada essa configuração inicial, algumas técnicas podem ser utilizadas para simplificar a resolução do problema, além de poder ser resolvida da melhor maneira possível, evitando atrasos ou possíveis transtornos.

Uma técnica se baseia na *restrição de domínio* do problema, a fim de reduzir a quantidade de células indeterminadas. Apenas essa técnica já é capaz de resolver os *puzzles* mais simples, mas não ainda assim ajuda na resolução. Para restringir o domínio são seguidos os seguintes passos:

1. Realizar anotação para cada célula quais números ainda não foram usados na linha, coluna e bloco em que se encontra. Esses são os números ainda possíveis para esta célula;
2. Preencher todas as células que possuem apenas um número possível;
3. Retornar ao passo 1 com o novo estado após o passo 2. Repetir até que não haja mais células com um único número possível.

Essa técnica remove do caminho todas as células óbvias do problema, permanecem vazias apenas as não determinísticas. Ainda sobre esse novo estado é possível usar da técnica chamada de *Emparelhamento*. O emparelhamento consiste em encontrar duas células em uma mesma linha que possuam os mesmos dois únicos números possíveis. Isso implica que nenhuma outra célula da linha pode conter esses dois números. Essa mesma lógica pode ser aplicada para

"*trincas'*, "*quadras*" ou mais, sendo limitados apenas pelo tamanho da grade. Aplicando o emparelhamento, é possível encontrar novas células óbvias. Então a restrição de domínio é aplicada novamente.

Alternar entre essas duas técnicas até que não possam mais ser aplicadas reduz muito o domínio de diversos *puzzles*, mas ainda é possível que restem células indeterminadas. Nesses casos resta apenas "*chutar"* números para cada célula dentro dos números possíveis restantes e tentar restringir novamente o domínio da nova grade. Se algum caso inválido surgir, devesse retornar para o último caso válido (*backtracking*), e tentar outros números.

Implementação

Como mencionado anteriormente, a implementação foi feita em Java. Inicialmente foi escrita uma classe que representa o jogo de Sudoku (*SudokuGame*) e que possui funções básicas, como ler o arquivo de entrada nos moldes da especificação do trabalho e traduzí-lo para estruturas manipuláveis pela linguagem, também funções para inicializar os valores da grade e para mostrar seu estado. Existe outra classe também para cuidar de funções para manipular as estruturas de dados presentes dentro do *SudokuGame*. Posteriormente, foi escrita uma outra classe responsável por conter toda a lógica necessária para resolver um jogo de Sudoku informado (*SudokuSolver*). Esta última possui uma função para resolver o jogo com uma busca cega e outra com busca heurística. Todas as outras funções associadas a checar a validade do estado atual do jogo (*checkLine*, *checkColunm* e *checkBlock*), para executar a restrição de domínio (*restrictDomain*) e emparelhamentos (*matchingProbs*) também estão nessa classe, além das funções auxiliares para executar todas essas tarefas.

Na ciência da computação, há diversas classes de problemas, porém, destacou-se a classe NP que é quando podemos verificar sua solução de modo eficiente, assim, é evidente que os humanos são bons em resolver problemas NP aproximadamente, e em reciprocidade, problemas que achamos interessantes (como o do Tetris) com frequência têm "algo de NP". Dessa forma, uma das definições dadas à inteligência artificial diz que sua função é encontrar soluções heurísticas para problemas NP-completos (Domingos, 2017).

Tanto para o método da busca heurística quanto para o da busca cega, utilizou-se a busca em largura na expansão dos estados do jogo. Expandir um nó é derivar outros jogos possíveis a partir do estado atual. As funções que trata de expandir e a que gerenciar os jogos derivados também está na classe *SudokuSolver*. Para seguir a lógica da busca em largura, cada novo estado expandido é adicionado no final de uma fila. A fila segue a ordem FILO (*First-in, Last-out*). Funciona da seguinte forma:

- Recebe o primeiro estado da fila;
- Busca a primeira célula não preenchida da esquerda para a direita e de cima para baixo;
- Testa todas os números possíveis para aquela célula;
- Adiciona no fim da fila apenas os que resultam em uma nova grade válida;
- Remove o estado atual da fila.

A cada iteração é verificado se o estado atual está completo e é válido. Caso seja, a função retorna o jogo completo. Existe uma pequena diferença na expansão dos estados entre a busca heurística e cega. Na busca heurística é priorizada a célula com o "*Menor domínio primeiro*", ou seja, que possui menos possibilidades de números para testar, a fim de diminuir o grau de expansão da árvore de filhos nos testes.

A utilização de métodos heurísticos com o intuito de otimizar processos ou solução de problemas diversos ocorre, normalmente, quando não existem alternativas viáveis, um exemplo é quando não é possível utilizarmos métodos baseados na determinação de derivadas (BISCAIA JUNIOR, 2014). Portanto, métodos heurísticos utilizados para resolver problemas de otimização discreta não obtêm sempre respostas viáveis, porém seu enfoque intuitivo permite que, a partir de interpretação e exploração adequadas, uma solução razoável possa ser obtida (ARENALES et al., 2015).

Dessa maneira, utilizou-se a solução *Replit* para a realização do desenvolvimento da implementação do jogo. O *Replit* é uma IDE *online*, onde possui a capacidade de disponibilizar apoio para desenvolver programas em diversas linguagens de programação, tendo como diferencial a simplicidade de funcionamento e telas intuitivas.

A seguir são apresentadas as linhas de codificação implementadas para o desenvolvimento dos métodos de busca, sendo separado em 4 arquivos específicos com suas devidas funcionalidades dentro do programa, sendo elas:

Listagem 1. Método Principal – Main.Java

```
1   import Sudoku.SudokuGame;
2   import Sudoku.SudokuSolver;
3
4   import java.util.ArrayList;
5   import java.util.Arrays;
6   import java.util.List;
7   import java.util.concurrent.TimeUnit;
8
9   class Main {
10    public static void main(String[] args) {
11      SudokuGame myGame = new SudokuGame("Samples/inputSample4.txt");
12      //myGame.showGameState();
13
14      SudokuSolver solver = new SudokuSolver(myGame);
15      long inicio, fim;
16      TimeUnit time = TimeUnit.MICROSECONDS;
17
18      inicio = System.nanoTime();
19      solver.heuristicSolver();
20      fim = System.nanoTime();
21
22      solver.heuristicGame.showGameState();
23      System.out.println((fim - inicio)/1000 + " ms");
24
25
26      inicio = System.nanoTime();
27      solver.blindSolver();
28      fim  = System.nanoTime();
29
30      solver.blindGame.showGameState();
31      System.out.println((fim - inicio)/1000 + " ms");
```

```
32
33      /*
34      boolean valid = solver.validGame(myGame);
35      System.out.println(valid);
36      */
37    }
38  }
```

Nesta listagem apresentou-se o método *main* da codificação do programa, sendo esse o principal método ao qual permite e oferecer apoio para os outros métodos aos quais serão discorridos ao longo do artigo.

Listagem 2. Classe de Arranje - ArrayHelper.class

```
1   package ArrayUp;
2
3   public class ArrayHelper{
4
5     public static int[] append(int[] arr, int element) {
6       int[] array = new int[arr.length + 1];
7       for(int i = 0; i < arr.length; i++){
8         array[i] = arr[i];
9       }
10      array[arr.length] = element;
11      return array;
12    }
13
14    public static int[] remove(int[] arr, int element) {
15      if(arr == null || (contain(arr, element) == 0)) return arr;
16
17      boolean hasRemoved = false;
18      int[] anotherArray = new int[arr.length - 1];
19
20      for(int i = 0, k = 0; i < arr.length; i++) {
21        if (arr[i] == element) {
22          hasRemoved = true;
23          continue;
24        }
25        anotherArray[k++] = arr[i];
26      }
27
28      return (hasRemoved) ? anotherArray : arr;
29    }
30
31    public static int contain(int[] arr, int element) {
32      int count_contained = 0;
33
34      for(int i = 0; i < arr.length; i++)
35        if(arr[i] == element) count_contained += 1;
```

```
36
37      return count_contained;
38    }
39
40    public static boolean equalArrays(int[] arr1, int[] arr2) {
41      if(arr1 == null || arr2 == null) return false;
42      if(arr1.length != arr2.length) return false;
43
44      boolean is_equals = true;
45      for(int i = 0; i < arr1.length; i++){
46        if(arr1[i] != arr2[i]) is_equals = false;
47      }
48
49      return is_equals;
50    }
51
52  }
```

O presente arquivo codificado permite que seja realizada a organização das variáveis e elementos aos quais serão utilizados dentro do programa, sendo uma classe importante para o funcionamento do mesmo.

Listagem 3. Codificação do Jogo – SudokuGame.Java

```
1   package Sudoku;
2
3   import java.io.File;
4   import java.io.FileNotFoundException;
5   import java.util.Scanner;
6   import java.util.Arrays;
7   import ArrayUp.ArrayHelper;
8
9   public class SudokuGame {
10
11  private String inputGamePath;
12  private String[] inputGame = new String[11];
13  public int[][] gameState = new int [9][0];
14  public int[][][] probabilities = new int [9][9][9];
15  public int gameLength = 9;
16
17  public SudokuGame(String gamePath){
18  if(gamePath != null){
19  inputGamePath = gamePath;
20  readGameFile();
21  translateInputGame();
22  } else {
23  initGameState();
24  }
```

```
25  initProbalities();
26  }
27
28  public int[][] getGameState(){
29  return this.gameState;
30  }
31
32  public int[][][] getProbabilities(){
33  return this.probabilities;
34  }
35
36  public void setGameState(int[][] gameState) {
37  if(gameState != null){
38  for(int i = 0; i < gameState.length; i++){
39  for(int j = 0; j < gameState[i].length; j++){
40  this.gameState[i][j] = gameState[i][j];
41  }
42  }
43  }
44  }
45
46  public void setProbabilities(int[][][] probabilities) {
47  if(probabilities != null){
48  for(int i = 0; i < probabilities.length; i++){
49  for(int j = 0; j < probabilities[i].length; j++){
50  if(probabilities[i][j] == null){
51  this.probabilities[i][j] = null;
52  } else {
53  for(int k = 0; k < probabilities[i][j].length; k++){
54  this.probabilities[i][j][k] = probabilities[i][j][k];
55  }
56  }
57  }
58  }
59  } else {
60  this.probabilities = null;
61  }
62  }
63
64  private void readGameFile(){
65  try{
66  File gameFile = new File(inputGamePath);
67  Scanner myReader = new Scanner(gameFile);
68  int count = 0;
69
70  while(myReader.hasNextLine()){
71  String data = myReader.nextLine();
72
73  inputGame[count] = data;
74  count++;
```

```
75      }
76
77      myReader close();
78      } catch(FileNotFoundException error){
79      System out println("As error ocurred!");
80      error printStackTrace();
81      }
82      }
83
84      private void translateInputGame(){
85      int countState = 0;
86      for(int i = 0; i < 11; i++){
87      if(inputGame[i] charAt(0) == '-') continue;
88
89      int[] getLineNumber = new int[0];
90
91      for(int j = 0; j < 21; j += 2){
92      if(inputGame[i] charAt(j) == '|') continue;
93      if(inputGame[i] charAt(j) == '_') {
94
95      getLineNumber = ArrayHelper append(getLineNumber, 0);
96      continue;
97      }
98      getLineNumber = ArrayHelper append(getLineNumber,
Character getNumericValue(inputGame[i] charAt(j)));
99      }
100
101     for(int element : getLineNumber){
102     gameState[countState] = ArrayHelper append(gameState[countState], element);
103     }
104
105     countState++;
106     }
107     }
108
109     private void initGameState(){
110     for(int i = 0; i < gameState length; i++){
111     int[] newLine = new int[9];
112     for(int j = 0; j < gameState length; j++){
113     newLine[j] = 0;
114     }
115     gameState[i] = newLine;
116     }
117     }
118
119     private void initProbalities(){
120     for(int i = 0; i < probabilities length; i++){
121     for(int j = 0; j < probabilities[i] length; j++){
122     for(int k = 0; k < probabilities[i][j] length; k++){
```

```
123     probabilities[i][j][k] = k+1;
124     }
125     }
126     }
127     }
128
129     public void showGameState(){
130     for(int[] arr : gameState){
131     System.out.println(Arrays.toString(arr));
132     }
133     System.out.println("--------------------------");
134     }
135
136     }
```

Após a codificação do método principal e criação da classe, a Listagem 3 se encarregará da montagem do jogo em si, atentando-se aos aspectos explicados anteriormente para o bom funcionamento do Jogo, sendo escalado em uma grade 9x9 separada em 9 blocos 3x3, para que assim, possa o solucionado possa desempenhar o seu papel de maneira eficaz e da melhor forma possível.

Listagem 4. Solucionador do Jogo – SudokuSolver.Java

```
1   package Sudoku;
2
3   import java.util.function.*;
4   import java.util.ArrayList;
5   import java.util.Arrays;
6   import java.util.Collections;
7   import java.util.List;
8
9   import Sudoku.SudokuGame;
10  import ArrayUp.ArrayHelper;
11
12  public class SudokuSolver {
13
14  public SudokuGame initGame;
15  public SudokuGame blindGame;
16  public SudokuGame heuristicGame;
17
18  public SudokuSolver(SudokuGame newGame) {
19  initGame = new SudokuGame(null);
20  initGame.setGameState(newGame.gameState);
21
22  blindGame = new SudokuGame(null);
23  blindGame.setGameState(newGame.gameState);
24  restrictDomain(blindGame);
25
26  heuristicGame = new SudokuGame(null);
```

```
heuristicGame.setGameState(newGame.gameState);
restrictDomain(heuristicGame);
}

public boolean heuristicSolver() {

//Recebe as coordenadas da célula com menor domínio
int[] coords = minDomain(heuristicGame);
int i = coords[0];
int j = coords[1];
//Se o valor for 10, é pq não existem células vazias
if(coords[0] == 10) return true;

if(heuristicGame gameState[i][j] == 0){
//Testa todos os valores possíveis para aquela célula
for(int k = 0; k < heuristicGame.probabilities[i][j].length; k++){
if(is_possible(i, j, heuristicGame.probabilities[i][j][k])){
heuristicGame.gameState[i][j] = k;
/*Recursivamente testa se o jogo já foi totalmente preenchido
E se continua válido. Caso sim, retorna true. Senão,
apenas preenche o valor com 0 novamente para testar outro número*/
if(validGame(heuristicGame) && heuristicSolver()) {
return true;
}
heuristicGame.gameState[i][j] = 0;
}
}
return false;
}

return true;
}

public boolean blindSolver() {
// Exata mesma lógica do heuristicSolver, porém não prioriza
// as células com menor domínio. Apenas testa todas da esquerda
// para a direita e de cima para baixo.
for(int i = 0; i < 9; i++){
for(int j = 0; j < 9; j++){
if(blindGame.gameState[i][j] == 0){
for(int k = 0; k <= 9; k++){
if(is_possible(i, j, k)){
blindGame.gameState[i][j] = k;
if(validGame(blindGame) && blindSolver()) {
return true;
}
blindGame.gameState[i][j] = 0;
}
}
return false;
```

```
77  }
78  }
79  }
80  return true;
81  }
82
83  public int[] minDomain(SudokuGame game){
84  int[] coords = {10, 10};
85  int size = 10;
86
87  //Busca a célula com menor domínio comparando os tamanhos
88  //dos arrays de probabilidades de cada célula.
89  for(int i = 0; i < 9; i++){
90  for(int j = 0; j < 9; j++){
91  if(game.probabilities[i][j] == null) continue;
92  if(game.probabilities[i][j].length < size){
93  coords[0] = i;
94  coords[1] = j;
95  size = game.probabilities[i][j].length;
96  }
97  }
98  }
99
100 return coords;
101 }
102
103 public boolean validGame(SudokuGame game) {
104 // line
105 for (int i = 0; i < game.probabilities.length; i++) {
106 for (int j = 0; j < game.probabilities[i].length; j++) {
107 if (game.gameState[i][j] != 0 && ArrayHelper.contain(game.gameState[i],
    game.gameState[i][j]) > 1) {
108 return false;
109 }
110 }
111 }
112
113 // column
114 for (int i = 0; i < game gameState length; i++) {
115 List<Integer> column = new ArrayList<Integer>();
116
117 for (int j = 0; j < game.gameState[i].length; j++) {
118 column.add(game.gameState[j][i]);
119 }
120 for (int j = 0; j < game.probabilities[i].length; j++) {
121 if (game.gameState[i][j] != 0 && Collections.frequency(column, game.gameState[i][j])
    > 1) {
122 return false;
123 }
124 }
```

```
125 }
126
127 // block
128 for (int i = 0; i < 9; i += 3) {
129 for (int j = 0; j < 9; j += 3) {
130 List<Integer> block = new ArrayList<Integer>();
131
132 for (int k = 0; k < 3; k++) {
133 for (int l = 0; l < 3; l++) {
134 block add(game gameState[k + i][l + j]);
135 }
136 }
137
138 for (int k = 0; k < game.probabilities.length; k++) {
139 for (int l = 0; l < game.probabilities[k].length; l++) {
140 if (game.gameState[k][l] != 0 && Collections.frequency(block, game.gameState[k][l])
    > 1) {
141 return false;
142 }
143 }
144 }
145
146 }
147 }
148
149 return true;
150 }
151
152 public boolean is_possible(int x, int y, int num){
153 // Busca saber se é possível colocar num na célula x e y
154 // sem invalidar o jogo.
155
156 for(int i = 0; i < 9; i++){
157 if(blindGame.gameState[x][i] == num) return false;
158 if(blindGame.gameState[i][y] == num) return false;
159 }
160
161 int i = setBlockCoordinate(x);
162 int j = setBlockCoordinate(y);
163
164 for(int k = 0 ;k < 3; k++){
165 for(int l = 0; l < 3; l++){
166 if(blindGame.gameState[k+i][j+l] == num) return false;
167 }
168 }
169
170 return true;
171 }
172
173 private void restrictDomain(SudokuGame game) {
```

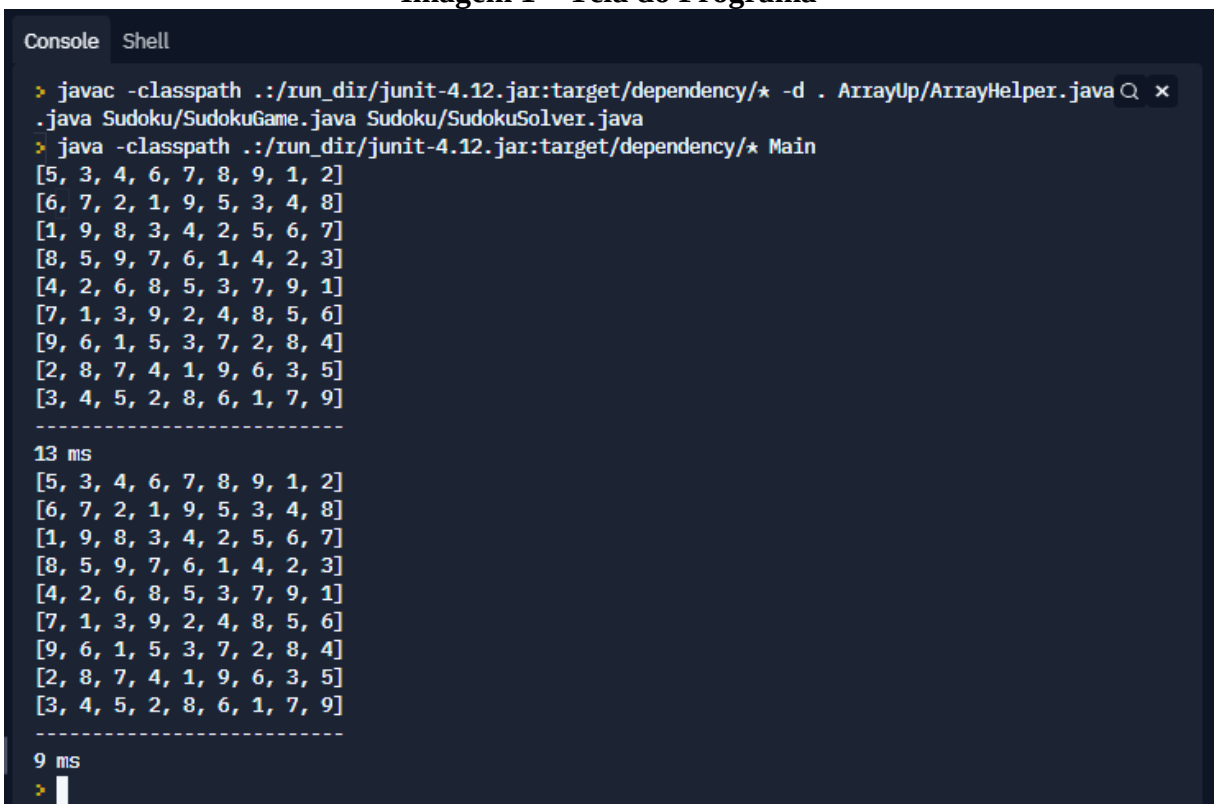
```
174 // Sempre que campos óbvio forem preenchidos, serão atualizadas
175 // as probabilidades de cada célula. Até que não haja mais campos
176 // óbvios.
177 do {
178 updateProbabilities(game);
179 } while (fillObviousFields(game));
180 }
181
182 private void updateProbabilities(SudokuGame game) {
183 for (int i = 0; i < game.probabilities.length; i++) {
184 for (int j = 0; j < game.probabilities[i].length; j++) {
185 if (game.gameState[i][j] == 0) {
186 // Atualiza os campos possíveis da célula i, j pela linha
187 game.probabilities[i][j] = checkLine(game.probabilities[i][j], i, game);
188 // Atualiza os campos possíveis da célula i, j pela coluna
189 game.probabilities[i][j] = checkColumn(game.probabilities[i][j], j, game);
190 // Atualiza os campos possíveis da célula i, j pelo bloco
191 game.probabilities[i][j] = checkBlock(game.probabilities[i][j], i, j, game);
192 } else {
193 game.probabilities[i][j] = null;
194 }
195 }
196 // Executa o emparelhamento em duplas na linha i
197 game.probabilities[i] = matchingProbs(game.probabilities[i], game);
198 }
199 }
200
201 private boolean fillObviousFields(SudokuGame game) {
202 boolean was_filled = false;

203 for (int i = 0; i < game.gameState.length; i++) {
204 for (int j = 0; j < game.gameState.length; j++) {
205 if (game probabilities[i][j] == null)
206 continue;
207 if (game.probabilities[i][j].length == 1) {
208 game.gameState[i][j] = game.probabilities[i][j][0];
209 game.probabilities[i][j] = null;
210 was_filled = true;
211 }
212 }
213 }
214
215 return was_filled;
216 }
217
218 private int[][] matchingProbs(int[][] line, SudokuGame game){
219 boolean finded = false;
220 for(int j = 0; j < line.length && !finded; j++){
221 if(line[j] == null || line[j].length != 2) continue;
222 for(int k = 0; k < line.length && !finded; k++){
```

```
223 if(k == j) continue;
224 if(ArrayHelper.equalArrays(line[j], line[k])){
225 line = resolveMatch(line, game, j, k);
226 finded = true;
227 }
228 }
229 }
230 return line;
231 }
232
233 private int[][] resolveMatch(int[][] line, SudokuGame game, int first, int second){
234 for(int i = 0; i < line length; i++){
235 if(i == first || i == second) continue;
236 for(int j = 0; j < line[first].length; j++){
237 line[i] = ArrayHelper.remove(line[i], line[first][j]);
238 }
239 }
240 return line;
241 }
242
243 private int[] checkLine(int[] line, int i, SudokuGame game) {
244 for (int k = 0; k < 9; k++) {
245 if (game.gameState[i][k] != 0) {
246 line = ArrayHelper.remove(line, game.gameState[i][k]);
247 }
248 }
249 return line;
250 }
251
252 private int[] checkColumn(int[] line, int j, SudokuGame game) {
253 for (int k = 0; k < 9; k++) {
254 if (game.gameState[k][j] != 0) {
255 line = ArrayHelper.remove(line, game.gameState[k][j]);
256 }
257 }
258 return line;
259 }
260
261 private int[] checkBlock(int[] line, int i, int j, SudokuGame game) {
262 i = setBlockCoordinate(i);
263 j = setBlockCoordinate(j);
264
265 for (int k = 0; k < 3; k++) {
266 for (int l = 0; l < 3; l++) {
267 if (game.gameState[k + i][l + j] != 0) {
268 line = ArrayHelper remove(line, game.gameState[k + i][l + j]);
269 }
270 }
271 }
272
```

```
273 return line;
274 }
275
276 private int setBlockCoordinate(int coordinate) {
277 int newCoordinate;
278
279 if (coordinate > 5)
280 newCoordinate = 6;
281 else if (coordinate > 2)
282 newCoordinate = 3;
283 else
284 newCoordinate = 0;
285
286 return newCoordinate;
287 }
288 }
```

Posteriormente, após ser realizada a criação do método principal, elaboração da classe e estruturação do jogo, será feita a codificação do solucionador, para que assim seja efetivada a resolução do jogo de maneira automatizada. Ademais, no decorrer da listagem 4 é descrita, por meio de comentários (//...), as principais linhas dentro da codificação, sendo considerada a explicação da função ao qual aquela linha comentada executa.

Imagem 1 – Tela do Programa



Após ser implementada a etapa de codificação e realizar a execução do programa, será apresentada a tela do programa ao qual disponibiliza visualmente o programa rodando. Ademais, caso haja algum problema em relação ao aparecimento da tela e a mesa não esteja

aparecendo, procure verificar o código e tentar identificar se não há algum erro, caso contrário verifique o próprio compilador, confirmando se não há algum *bug* ou algo do tipo.

Conclusão

Neste trabalho realizou-se a discussão referente a uma formulação de restrição do quebra-cabeça Sudoku, assim, viu-se que pode ser utilizada diferentes técnicas para encontrar soluções deste passatempo. As restrições adicionais totalmente diferentes nos blocos principais dão mais chances de encontrar restrições redundantes que possam ajudar no processo de solução, assim, facilitando um processo de resolução de puzzle e contribuindo para a eficácia do método de busca.

Os Quebra-cabeças Sudoku não são apenas uma adição interessante ao conjunto de problemas para desenvolver técnicas de restrição, mas também fornecem uma oportunidade única para atrair mais pessoas interessadas em programação desta modalidade. Além disso, pode contribuir positivamente para a desenvoltura do raciocínio lógico de seus usuários e ajudar na capacidade de memorização.

Referências

ARENALES, M. et al. *Pesquisa Operacional*. 2. ed. Rio de Janeiro: Campus, 2015.

BISCAIA JUNIOR, E. C. *Métodos Não Determinísticos*. Rio de Janeiro: UFRJ, 2014. Disponível em: <http://www2.peq.coppe.ufrj.br/Pessoal/Professores/Evaristo/CO897_2014/M%E9todos%20N%E3o%20Determin%EDsticos/aula1.pdf>.

Domingos, P. (2017). *O Algoritmo Mestre - Como a busca pelo algoritmo de machine learning definitivo recriará nosso mundo.* São paulo: Novatec Editora Ltda

RENATO, S. *Confira as 20 linguagens de programação mais populares do momento*. 2020. Disponível em: https://olhardigital.com.br/2020/03/03/pro/confira-as-20-linguagens-de-programacao-mais-populares-do-momento/.