Relatório da Implementação de Monte Carlo
Grupo:
Julian Degutis
William Henrique
Letícia Leite

1. Introdução
Deverá ser criado uma solução, em que, dado um tabuleiro ***nxn*** deverá ser otimizado o número de rainhas posicionadas em uma casa ***An***, que satisfaça as seguintes condições:

1. Cada rainha não deve atacar qualquer outra rainha do tabuleiro
2. Apenas uma rainha pode ser posicionada por casa.

2. Solução

Será utilizada uma implementação utilizando a linguagem Java, que terá como argumento em ordem:

- O número N representando o tabuleiro NxN.
- O número de iterações que o algoritmo de Monte Carlo deverá executar

O sistema, deverá calcular todas as diagonais da matriz, utilizando a técnica de pontos candidatos para o início de cada diagonal, sendo:

- Orientação para direita
  - Todos os pontos ***aij*** começando de ***a0,0*** até ***a0,n***
- Orientação para baixo
  - Todos os pontos ***aij*** começando de ***a1,0*** até ***an,0***
- Orientação para esquerda
  - Todos os pontos ***aij*** começando de ***an,n*** até ***a0,n***
- Orientação para cima
  - Todos os pontos ***aij*** começando de ***an-1,n-1*** até ***an,0***

Com posse de todas as linhas, colunas e diagonais, podemos tomar como solução que, uma rainha não ataca quaisquer outras rainhas distribuídas no tabuleiro, caso a soma de todas as rainhas de uma linha, uma coluna ou uma diagonal é menor ou igual a 1.

3. Implementação

Após a verificação das diagonais da matriz ***nxn,*** a implementação da solução é dada pelo seguinte algoritmo baseado no método de Monte Carlo:

1. FOR(ITERAÇÃO < PARÂMETRO MÁXIMO DE ITERAÇÃO)
    1. PREENCHE A MATRIZ NXN COM ZEROS
    2. RANDOMIZA DOIS NUMEROS PARA A POSIÇÃO INICIAL DA RAINHA
    3. ADICIONA A RAINHA NA POSIÇÃO X,Y DA MATRIZ
    4. PREENCHE A POSIÇÃO COMO UTILIZADA E PREENCHIDA
        1. ENQUANTO TODOS OS PONTOS NÃO FOREM TESTADOS
            1. RANDOMIZA DOIS NUMEROS PARA A POSIÇÃO DA NOVA RAINHA
            2. VERIFICA SE ELE JÁ FOI TESTADO ANTERIORMENTE
                1. NÃO
                    1. ADICIONA NA LISTA DE PONTOS TESTADOS
                    2. POSICIONA A RAINHA NA POSIÇÃO X,Y SORTEADA
                    3. VERIFICA A REGRA DO PROBLEMA DAS RAINHAS
                        1. POSIÇÃO OK
                            1. ADICIONA O PONTO NA LISTA SOLUÇÃO
                            2. ATUALIZA O VALOR DO ÓTIMO
                        2. POSIÇÃO NOK
                            1. REMOVE A RAINHA
    5. VERIFICA O ÓTIMO DA ITERAÇÃO CORRENTE COM A ÚLTIMA ITERAÇÃO
        1. ÓTIMO ANTIGO < ÓTIMO ATUAL
            1. ATUALIZA A LISTA DE SOLUÇÃO
            2. ATUALIZA O VALOR DO ÓTIMO FINAL

4. Casos de teste

Os testes foram realizados todos com 100 iterações do algoritmo do Monte Carlo para testar o ótimo do problema. Exemplo de execução da solução para um tabuleiro 8x8

Iniciando o processo da otimização do problema das rainhas em um tabuleiro de xadrez 8x8

Ótimo encontrado para iteração de número 0: 6
Ótimo encontrado para iteração de número 1: 7
…
Ótimo encontrado para iteração de número 28: 8
...
Ótimo encontrado para iteração de número 98: 6
Ótimo encontrado para iteração de número 99: 6

Ótimo após 100 interações: 8

Solução:
Ponto:1,7
Ponto:3,2
Ponto:4,8
Ponto:2,4
Ponto:8,5
Ponto:6,1
Ponto:7,3
Ponto:5,6

Tempo de execução: 64 milisegundos.
Aproximação de quantidade de memória gasta: -5452840 bytes.

Em comparação com o algoritmo do Simplex, tempos

N	Tempo MC (ms)	Memória MC (bytes)	Tempo Simplex (ms)	Memória Simplex (bytes)
8	64	5.452.840	35	2.728.376
20	482	9.807.480	148	7.650.776
90	136.032	17.905.840	39.547	256.061.544

```java
package montecarlo;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class MonteCarlo {

    private static String report;

    public static void main( String[] args ) {

        try {

            Integer size = Integer.parseInt( args[0] );
            Integer qntAttempt = Integer.parseInt( args[1] );
            long startTime = System.nanoTime();
            long beforeUsedMem =
            Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory();

            report = "Iniciando o processo da otimização do problema das rainhas em um
            tabuleiro de xadrez " + size + "x" + size + "\n\n";


            Integer M[][] = new Integer[size][size];

            List< List< Spot > > diagonals = new ArrayList< List< Spot > >();

            rightDiagonals( size, diagonals );
            leftDiagonals( size, diagonals );

            monteCarloAlgorithm( M, size, diagonals, qntAttempt );

            long endTime = System.nanoTime();
            long afterUsedMem = Runtime.getRuntime().totalMemory() -
            Runtime.getRuntime().freeMemory();

            report = report.concat( "Tempo de execução: " + ( ( endTime - startTime ) /
            1000000 ) + " milisegundos. \n" );
            report = report.concat( "Aproximação de quantidade de memória gasta: " + (
            beforeUsedMem - afterUsedMem ) + " bytes. \n" );

            FileWriter writer = new FileWriter( "report/MONTE_CARLO" + size + "x" +
            size +"_" + System.currentTimeMillis() + ".txt" );
            BufferedWriter bw = new BufferedWriter( writer );

            bw.write( report );

            bw.close();

        } catch( Exception ex ) {
            ex.printStackTrace();
        }

    }

    /**
     * Fills the matrix with zeros
     *
     * @param M
     * @param size
     */
    public static void fill( Integer M[][], Integer size ) {
        for( int i = 0 ; i < size ; i++ ) {
            for( int j = 0 ; j < size ; j++ ) {
                M[i][j] = 0;
            }
```

```
64          }
65      }
66
67      /**
68       *
69       * Process the Monte Carlo Algorithm
70       *
71       * @param M
72       * @param size
73       * @param diagonals
74       * @param qntAttempt
75       */
76      public static void monteCarloAlgorithm( Integer M[][], Integer size, List< List<
        Spot > > diagonals, Integer qntAttempt ) {
77
78          /**
79           * Best optimum of all iterations
80           */
81          Integer finalOptimum = 0;
82          List< Spot > solution = new ArrayList< Spot >();
83
84          for( int i = 0 ; i < qntAttempt ; i++ ) {
85
86              /**
87               * Reset the matrix with zeros
88               */
89              fill( M, size );
90
91              List< Spot > spotsTaken = new ArrayList< Spot >();
92              List< Spot > validSpots = new ArrayList< Spot >();
93              Integer optimum;
94              /**
95               * Take a random position and set into the matrix
96               */
97              Integer x = new Random().nextInt( size );
98              Integer y = new Random().nextInt( size );
99              spotsTaken.add( new Spot( x, y ) );
100             validSpots.add( new Spot( x, y ) );
101             M[x][y] = 1;
102             optimum = 1;
103
104             for(;;) {
105                 Integer tmpX = new Random().nextInt( size );
106                 Integer tmpY = new Random().nextInt( size );
107                 Spot tmpSpot = new Spot( tmpX, tmpY );
108                 /**
109                  * Checks if the spot is already taken by any other queen
110                  */
111                 if( !spotsTaken.contains( tmpSpot ) ) {
112                     spotsTaken.add( tmpSpot );
113                     /**
114                      * Puts the queen in the random spot
115                      */
116                     M[tmpX][tmpY] = 1;
117
118                     /**
119                      * Checks if its break the problem rule
120                      * Rule:
121                      * If breaks -> Take the queen of the temporary spot
122                      * If doesn't -> Increments the optimum
123                      */
124                     if( !validate( M, size, diagonals ) ) {
125                         M[tmpX][tmpY] = 0;
126                     } else {
127                         validSpots.add( new Spot( tmpX, tmpY ) );
128                         optimum++;
129                     }
130                 }
131                 /**
```

```
132                  * Do it until all spots are tried
133                  */
134                 if( spotsTaken.size() == size * size  ) {
135                     break;
136                 }
137             }
138 
139             /**
140              * Prints the optimum for current iteration
141              */
142             report = report.concat( "Ótimo encontrado para iteração de número " + i +
                ": " + optimum + "\n" );
143 
144             /**
145              * Checks if it is the best optimum
146              */
147             if( finalOptimum < optimum ) {
148                 solution = validSpots;
149                 finalOptimum = optimum;
150             }
151 
152         }
153 
154         report = report.concat( "\n" );
155 
156         report = report.concat( "Ótimo após " + qntAttempt + " interações: " +
            finalOptimum + "\n\n" );
157         report = report.concat( "Solução:\n" );
158         for( Spot spot : solution ) {
159             report = report.concat( "Ponto:" + ( spot.getX() + 1 ) + "," + (
                spot.getY() + 1 ) + "\n" );
160         }
161 
162     }
163 
164     /**
165      *
166      * Validates the queen rules for this problem
167      *
168      * @param M
169      * @param size
170      * @param diagonals
171      * @return False if the rule is broke / True if the rules continues
172      */
173     public static Boolean validate( Integer M[][], Integer size, List< List< Spot > >
        diagonals ) {
174 
175         /**
176          * Checking line/column
177          */
178         for( int i = 0 ; i < size ; i++ ) {
179             Integer validatorColumn = 0;
180             Integer validatorLine = 0;
181             for( int j = 0 ; j < size ; j++ ) {
182                 validatorColumn = validatorColumn + M[i][j];
183                 validatorLine = validatorLine + M[j][i];
184                 if( validatorColumn > 1 || validatorLine > 1 ) {
185                     return Boolean.FALSE;
186                 }
187             }
188         }
189 
190         /**
191          * Checking diagonals
192          */
193         for( List< Spot > spots : diagonals ) {
194             Integer validatorDiagonal = 0;
195             for( Spot spot : spots ) {
196                 validatorDiagonal = validatorDiagonal + M[spot.getX()][spot.getY()];
```

```
                if( validatorDiagonal > 1 ) {
                    return Boolean.FALSE;
                }
            }
        }

        return Boolean.TRUE;
    }

    /**
     *
     * Find all left oriented diagonais from a N (as size) dimensional matrix
     *
     * @param M
     * @param size
     * @param diagonals
     */
    public static void leftDiagonals( Integer size, List< List< Spot > > diagonals ) {
        /**
         * Column iteraction
         */
        for( int i = 1 ; i <= size ; i++ ) {
            Integer tmpLine = i;
            Integer tmpColumn = size;
            List< Spot > currentDiagonal = new ArrayList< Spot >();
            while( tmpLine <= size && tmpColumn <= size ) {
                currentDiagonal.add( new Spot( tmpLine - 1, tmpColumn - 1 ) );
                tmpLine++;
                tmpColumn--;
            }
            if( currentDiagonal.size() > 1 ) {
                diagonals.add( currentDiagonal );
            }
        }

        /**
         * Line iteraction
         */
        for( int i = size - 1 ; i >= 1 ; i-- ) {
            Integer tmpLine = 1;
            Integer tmpColumn = i;
            List< Spot > currentDiagonal = new ArrayList< Spot >();
            while( tmpLine <= size && tmpColumn >= 1 ) {
                currentDiagonal.add( new Spot( tmpLine - 1, tmpColumn - 1 ) );
                tmpLine++;
                tmpColumn--;
            }
            if( currentDiagonal.size() > 1 ) {
                diagonals.add( currentDiagonal );
            }
        }
    }

    /**
     *
     * Find all right oriented diagonals from a N (as size) dimensional matrix
     *
     * @param M
     * @param size
     * @param diagonals
     */
    public static void rightDiagonals( Integer size, List< List< Spot > > diagonals ) {

        /**
         * Column iteraction
         */
        for( int i = 1 ; i <= size ; i++ ) {
            Integer tmpLine = i;
            Integer tmpColumn = 1;
```

```
266             List< Spot > currentDiagonal = new ArrayList< Spot >();
267             while( tmpLine <= size && tmpLine <= size ) {
268                 currentDiagonal.add( new Spot( tmpLine - 1, tmpColumn - 1 ) );
269                 tmpLine++;
270                 tmpColumn++;
271             }
272             if( currentDiagonal.size() > 1 ) {
273                 diagonals.add( currentDiagonal );
274             }
275         }
276 
277         /**
278          * Line iteraction
279          */
280         for( int i = 2 ; i <= size ; i++ ) {
281             Integer tmpLine = 1;
282             Integer tmpColumn = i;
283             List< Spot > currentDiagonal = new ArrayList< Spot >();
284             while( tmpLine <= size && tmpColumn <= size ) {
285                 currentDiagonal.add( new Spot( tmpLine - 1, tmpColumn - 1 ) );
286                 tmpLine++;
287                 tmpColumn++;
288             }
289             if( currentDiagonal.size() > 1 ) {
290                 diagonals.add( currentDiagonal );
291             }
292         }
293     }
294 
295 }
296 
```

```java
package montecarlo;

import java.io.Serializable;

public class Spot implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 18762942213554653O5L;

    private Integer x;

    private Integer y;

    public Spot( Integer x, Integer y ) {
        this.x = x;
        this.y = y;
    }

    public Integer getX() {
        return x;
    }

    public void setX(Integer x) {
        this.x = x;
    }

    public Integer getY() {
        return y;
    }

    public void setY(Integer y) {
        this.y = y;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((x == null) ? 0 : x.hashCode());
        result = prime * result + ((y == null) ? 0 : y.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Spot other = (Spot) obj;
        if (x == null) {
            if (other.x != null)
                return false;
        } else if (!x.equals(other.x))
            return false;
        if (y == null) {
            if (other.y != null)
                return false;
        } else if (!y.equals(other.y))
            return false;
        return true;
    }

}
```