Relatório da solução do problema das rainhas.
Grupo:
Julian Degutis de Freitas Garcia
Letícia Leite Caetano
William Henrique

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 o número N representando o tabuleiro NxN.
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.

Criando as restrições para a maximização da soma de todas as posições do tabuleiro, podemos criar um modelo de programação linear para o algoritmo do Simplex solucionar.

Na solução, será necessário tornar a matriz representativa do tabuleiro, em um vetor, para que consigamos atribuir o valor zerado de cada variável que não será utilizada em cada uma das restrições.

Para isso, será utilizada a implementação da biblioteca SCPSolver.jar. No primeiro momento, tentamos utilizar a solução da Apache, porém sem sucesso para valores de variáveis booleanas, por isso modificamos a abordagem da solução.

As bibliotecas para a execução do programa se encontram em:
https://github.com/juliandegutis/simplex/tree/master/lib

A solução e modelagem é apresentada na pasta /report/* da raiz do projeto, contento a modelagem em sí, a solução ótima, os valores das variáveis, o tempo de execução e a aproximação da memória gasta pelo sistema.

3. Casos de teste

Foram realizados 11 casos de testes, para um tabuleiro normal 8x8 e todos demais tabuleiros múltiplos de 9. Toda a solução se encontra em anexo ao e-mail. Passaremos brevemente os casos de teste nesse documento. Note que a memória é apenas uma estimação realizada pelo sistema, sendo (quantidade de memória disponível antes e após a execução do processo), os valores podem não ser 100% preciso, devidos aos processos existentes rodando no computador.

Valor de n	Solução Ótima	Tempo (milisegundos)	Memória (bytes)
8	8	35	2728376
9	9	32	3409872
18	18	97	16501016
27	27	327	40419056
36	36	785	227733664
45	45	2217	311807928
54	54	3070	359156080
63	63	7174	100543504
72	72	11598	387024936
81	81	25205	104154432
90	90	39547	256061544

3.1 Exemplo de solução para um tabuleiro 5x5

O log de execução com a modelagem gerada para o problema, se encontra na pasta /reports/ para cada execução realizada. Exemplificando, segue a execução gerada para um tabuleiro com n = 5

Iniciando o processo da otimização do problema das rainhas em um tabuleiro de xadrez 5x5

max z = x1,1 + x1,2 + x1,3 + x1,4 + x1,5 + x2,1 + x2,2 + x2,3 + x2,4 + x2,5 + x3,1 + x3,2 + x3,3 + x3,4 + x3,5 + x4,1 + x4,2 + x4,3 + x4,4 + x4,5 + x5,1 + x5,2 + x5,3 + x5,4 + x5,5

s.a:

x1,1 + x1,2 + x1,3 + x1,4 + x1,5 <= 1

x2,1 + x2,2 + x2,3 + x2,4 + x2,5 <= 1

x3,1 + x3,2 + x3,3 + x3,4 + x3,5 <= 1

x4,1 + x4,2 + x4,3 + x4,4 + x4,5 <= 1

$x_{5,1} + x_{5,2} + x_{5,3} + x_{5,4} + x_{5,5} \leq 1$

$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} + x_{5,1} \leq 1$

$x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} + x_{5,2} \leq 1$

$x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} + x_{5,3} \leq 1$

$x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} + x_{5,4} \leq 1$

$x_{1,5} + x_{2,5} + x_{3,5} + x_{4,5} + x_{5,5} \leq 1$

$x_{1,1} + x_{2,2} + x_{3,3} + x_{4,4} + x_{5,5} \leq 1$

$x_{2,1} + x_{3,2} + x_{4,3} + x_{5,4} \leq 1$

$x_{3,1} + x_{4,2} + x_{5,3} \leq 1$

$x_{4,1} + x_{5,2} \leq 1$

$x_{1,2} + x_{2,3} + x_{3,4} + x_{4,5} \leq 1$

$x_{1,3} + x_{2,4} + x_{3,5} \leq 1$

$x_{1,4} + x_{2,5} \leq 1$

$x_{1,5} + x_{2,4} + x_{3,3} + x_{4,2} + x_{5,1} \leq 1$

$x_{2,5} + x_{3,4} + x_{4,3} + x_{5,2} \leq 1$

$x_{3,5} + x_{4,4} + x_{5,3} \leq 1$

$x_{4,5} + x_{5,4} \leq 1$

$x_{1,4} + x_{2,3} + x_{3,2} + x_{4,1} \leq 1$

$x_{1,3} + x_{2,2} + x_{3,1} \leq 1$

$x_{1,2} + x_{2,1} \leq 1$

$x_{i,j}, 1 \leq i \leq 5, 1 \leq j \leq 5 \in \{ 0, 1 \}$

Posição 5 - 1.0
Posição 7 - 1.0
Posição 14 - 1.0
Posição 16 - 1.0
Posição 23 - 1.0
Solução ótima: 5 rainhas posicionadas no tabuleiro.

Tempo de execução: 90 milisegundos.
Aproximação de quantidade de memória gasta: -681784 bytes.

```java
package simplex;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.commons.math3.optim.PointValuePair;
import org.apache.commons.math3.optim.linear.LinearConstraint;
import org.apache.commons.math3.optim.linear.LinearConstraintSet;
import org.apache.commons.math3.optim.linear.LinearObjectiveFunction;
import org.apache.commons.math3.optim.linear.NonNegativeConstraint;
import org.apache.commons.math3.optim.linear.PivotSelectionRule;
import org.apache.commons.math3.optim.linear.Relationship;
import org.apache.commons.math3.optim.linear.SimplexSolver;
import org.apache.commons.math3.optim.nonlinear.scalar.GoalType;

import scpsolver.constraints.LinearSmallerThanEqualsConstraint;
import scpsolver.lpsolver.LinearProgramSolver;
import scpsolver.lpsolver.SolverFactory;
import scpsolver.problems.LinearProgram;

public class SimplexMain {

    private static String report;

    public static void main( String[] args ) {

        try {

            Integer size = Integer.parseInt( args[0] );
            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";

            /**
             * Object to map the M[í][j] point in a one-dimensional matrix
             */
            Map< String, Integer > mapPosition = new HashMap< String, Integer >();
            fill( mapPosition, size );

            Integer M[][] = new Integer[size - 1][size - 1];

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

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

            Tableau tableau = buildModel( size, diagonals, mapPosition );

            optimizeBool( tableau, size );

            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/MODEL_" + size + "x" + size
            +"_" + System.currentTimeMillis() + ".txt" );
            BufferedWriter bw = new BufferedWriter( writer );
```

```
64
65              bw.write( report );
66
67              bw.close();
68
69          } catch( Exception ex ) {
70              ex.printStackTrace();
71          }
72
73      }
74
75      public static void fill( Map< String, Integer > mapPosition, Integer size ) {
76
77          Integer pos = 0;
78          for( int i = 1 ; i <= size ; i++ ) {
79              for( int j = 1 ; j <= size ; j++ ) {
80                  String variable = "x" + i + "," + j;
81                  mapPosition.put( variable, pos );
82                  pos++;
83              }
84          }
85
86      }
87
88      /**
89       *
90       * Optimize the current tableau with boolean variables
91       *
92       * @param tableau
93       * @param size
94       */
95      public static void optimizeBool( Tableau tableau, Integer size ) {
96
97          try {
98
99              LinearProgram lp = new LinearProgram( tableau.getZFunction() );
100
101             for( int i = 0 ; i < size * size ; i++ ) {
102                 lp.setBinary( i );
103             }
104
105             lp.setMinProblem( false );
106
107             Integer identifier = 1;
108             for( double[] restriction : tableau.getRestrictions() ) {
109                 lp.addConstraint(new LinearSmallerThanEqualsConstraint( restriction,
                    1.0, (identifier++).toString() ) );
110             }
111
112             LinearProgramSolver solver  = SolverFactory.newDefault();
113             double[] sol = solver.solve( lp );
114
115             Long optimum = 0L;
116             for( int i = 0 ; i < sol.length ; i++ ) {
117                 if( sol[i] == 1.0 ) {
118                     optimum++;
119                     report = report.concat( "Posição " + ( i + 1 ) + " - " + sol[i] +
                        "\n" );
120                 }
121             }
122
123             System.out.println( "OPTIMUM   " + optimum );
124
125             report = report.concat( "Solução ótima: " + optimum + " rainhas
                posicionadas no tabuleiro. \n" );
126
127         } catch( Exception ex ) {
128             ex.printStackTrace();
129         }
```

```
130
131
132         }
133
134         /**
135          * Not usable yet
136          * @param tableau
137          * @param size
138          */
139         public static void optimizeReal( Tableau tableau, Integer size ) {
140
141             try {
142
143                 LinearObjectiveFunction zFunction = new LinearObjectiveFunction(
                    tableau.getZFunction(), 0 );
144                 ArrayList< LinearConstraint > constraints = new
                    ArrayList<LinearConstraint>();
145                 for( double[] restriction : tableau.getRestrictions() ) {
146                     constraints.add( new LinearConstraint( restriction, Relationship.LEQ, 1
                        ) );
147                 }
148
149                 SimplexSolver solver = new SimplexSolver();
150
151                 PointValuePair solution = solver.optimize(zFunction, new
                    LinearConstraintSet(constraints),
152                         GoalType.MAXIMIZE,
153                         new NonNegativeConstraint(true),
154                         PivotSelectionRule.BLAND);
155
156             } catch( Exception ex ) {
157                 ex.printStackTrace();
158             }
159
160         }
161
162         /**
163          *
164          * Build and print the current model
165          *
166          * @param size
167          * @param diagonals
168          */
169         public static Tableau buildModel( Integer size, List< List< String > > diagonals,
            Map< String, Integer > mapPosition ) {
170
171             Tableau tableau = new Tableau();
172
173             /**
174              * Z-Function
175              */
176             String zFunction = "";
177             double[] zCoefs = new double[(size * size)];
178             int currZ = 0;
179             for( int i = 1 ; i <= size ; i++ ) {
180                 for( int j = 1 ; j <= size ; j++ ) {
181                     String variable = "x" + i + "," + j;
182                     zFunction = zFunction.concat( variable );
183                     zFunction = zFunction.concat( " + " );
184                     zCoefs[currZ++] = 1;
185                 }
186             }
187             tableau.setZFunction( zCoefs );
188
189             report = report.concat("max z = " + zFunction.substring(0, zFunction.length() -
                2) + "\n" );
190             report = report.concat( "\n" );
191             report = report.concat( "s.a: \n" );
192
```

```
193         /**
194          * Line restriction
195          */
196         for( int i = 1 ; i <= size ; i++ ) {
197             double[] restriction = new double[(size * size)];
198             report = report.concat( "\n" );
199             for( int j = 1 ; j <= size ; j++ ) {
200                 String variable = "x" + i + "," + j;
201                 report = report.concat( variable );
202                 if( j < size ) {
203                     report = report.concat( " + " );
204                 }
205                 restriction[ mapPosition.get( variable ) ] = 1;
206             }
207             tableau.addRestriction( restriction );
208             report = report.concat( " <= 1 \n" );
209         }
210 
211         /**
212          * Column restriction
213          */
214         for( int i = 1 ; i <= size ; i++ ) {
215             double[] restriction = new double[(size * size)];
216             report = report.concat( "\n" );
217             for( int j = 1 ; j <= size ; j++ ) {
218                 String variable = "x" + j + "," + i;
219                 report = report.concat( variable );
220                 if( j < size ) {
221                     report = report.concat( " + " );
222                 }
223                 restriction[ mapPosition.get( variable ) ] = 1;
224             }
225             tableau.addRestriction( restriction );
226             report = report.concat( " <= 1 \n" );
227         }
228 
229         /**
230          * Diagonal restriction
231          */
232         for( List< String > list : diagonals ) {
233             report = report.concat( "\n" );
234             double[] restriction = new double[(size * size)];
235             for( int i = 0 ; i < list.size() ; i++ ) {
236                 report = report.concat( "x" + list.get( i ) );
237                 if( i < list.size() - 1 ) {
238                     report = report.concat( " + " );
239                 }
240                 restriction[ mapPosition.get( "x" + list.get( i ) ) ] = 1;
241             }
242             tableau.addRestriction( restriction );
243             report = report.concat( " <= 1 \n" );
244         }
245 
246         /**
247          * Positive restriction
248          */
249         report = report.concat( " \n" );
250         report = report.concat( "xi,j, 1 <= i <= " + size + ", 1 <= j <= " + size + " E
        { 0, 1 } \n\n");
251 
252         return tableau;
253 
254     }
255 
256     /**
257      *
258      * Find all left oriented diagonais from a N (as size) dimensional matrix
259      *
260      * @param M
```

```
261         * @param size
262         * @param diagonals
263         */
264        public static void leftDiagonals( Integer M[][], Integer size, List< List< String >
           > diagonals ) {
265            /**
266             * Column iteraction
267             */
268            for( int i = 1 ; i <= size ; i++ ) {
269                Integer tmpLine = i;
270                Integer tmpColumn = size;
271                List< String > currentDiagonal = new ArrayList< String >();
272                while( tmpLine <= size && tmpColumn <= size ) {
273                    currentDiagonal.add( tmpLine + "," + tmpColumn );
274                    tmpLine++;
275                    tmpColumn--;
276                }
277                if( currentDiagonal.size() > 1 ) {
278                    diagonals.add( currentDiagonal );
279                }
280            }
281
282            /**
283             * Line iteraction
284             */
285            for( int i = size - 1 ; i >= 1 ; i-- ) {
286                Integer tmpLine = 1;
287                Integer tmpColumn = i;
288                List< String > currentDiagonal = new ArrayList< String >();
289                while( tmpLine <= size && tmpColumn >= 1 ) {
290                    currentDiagonal.add( tmpLine + "," + tmpColumn );
291                    tmpLine++;
292                    tmpColumn--;
293                }
294                if( currentDiagonal.size() > 1 ) {
295                    diagonals.add( currentDiagonal );
296                }
297            }
298        }
299
300        /**
301         *
302         * Find all right oriented diagonals from a N (as size) dimensional matrix
303         *
304         * @param M
305         * @param size
306         * @param diagonals
307         */
308        public static void rightDiagonals( Integer M[][], Integer size, List< List< String
           > > diagonals ) {
309
310            /**
311             * Column iteraction
312             */
313            for( int i = 1 ; i <= size ; i++ ) {
314                Integer tmpLine = i;
315                Integer tmpColumn = 1;
316                List< String > currentDiagonal = new ArrayList< String >();
317                while( tmpLine <= size && tmpLine <= size ) {
318                    currentDiagonal.add( tmpLine + "," + tmpColumn );
319                    tmpLine++;
320                    tmpColumn++;
321                }
322                if( currentDiagonal.size() > 1 ) {
323                    diagonals.add( currentDiagonal );
324                }
325            }
326
327            /**
```

```
328             * Line iteraction
329             */
330            for( int i = 2 ; i <= size ; i++ ) {
331                Integer tmpLine = 1;
332                Integer tmpColumn = i;
333                List< String > currentDiagonal = new ArrayList< String >();
334                while( tmpLine <= size && tmpColumn <= size ) {
335                    currentDiagonal.add( tmpLine + "," + tmpColumn );
336                    tmpLine++;
337                    tmpColumn++;
338                }
339                if( currentDiagonal.size() > 1 ) {
340                    diagonals.add( currentDiagonal );
341                }
342            }
343        }
344
345    }
346
```

```java
package simplex;

import java.util.LinkedList;
import java.util.List;

public class Tableau {

    private double[] zFunction;

    private List< double[] > restrictions;

    public Tableau() {}

    public double[] getZFunction() {
        return zFunction;
    }

    public void setZFunction( double[] zFunction ) {
        this.zFunction = zFunction;
    }

    public List< double[] > getRestrictions() {
        return restrictions;
    }

    public void setRestrictions(List< double[] > restrictions) {
        this.restrictions = restrictions;
    }

    public void addRestriction( double[] restriction ) {
        if( this.restrictions == null ) {
            this.restrictions = new LinkedList< double[] >();
        }
        this.restrictions.add( restriction );
    }

}

```