INSTITUTO FEDERAL CATARINENSE

NATALIA KELIM THIEL

ALGORITMO GENÉTICO:
IMPLEMENTAÇÃO DO JOGO 2048

RIO DO SUL

OUTUBRO/2018

INTRODUÇÃO

Inspirados na evolução, algoritmos genéticos são um conjunto de modelos computacionais que incorporam uma potencial solução à um problema específico numa estrutura cromossômica, aplicando operadores de seleção, reprodução e mutação. O alfabeto, que são os elementos na representação do cromossomo, podem ser de três formas: binária, inteira ou real. Cada cromossomo contém um conjunto de gene, estruturado pelo formato do alfabeto, que será passado por várias gerações a fim de otimizar e melhorar os resultados.

Inicialmente é gerada uma população de cromossomos aleatória para compor a geração zero. Então é aplicada uma avaliação, onde um fitness é gerado com base nas características do cromossomo. Quanto maior o fitness, melhor é este indivíduo. O método de seleção é utilizado na tentativa de juntar pares com os melhores fitness para reprodução. A reprodução combina os pares, resultando em dois filhos com suas características. Por fim é aplicado uma taxa de mutação a cada gene, que geralmente não passa de 0,5%, para simular a mutação natural.

Com este processo as gerações seguintes se tornam mais otimizadas e tendem a ser melhores. Entretanto cada execução pode trazer um resultado diferente, visto a aleatoriedade de seleção e possível mutação.

PROBLEMA

Este trabalho busca um algoritmo genético para resolver o jogo 2048, que tem como objetivo juntar pares de números iguais a fim de os soma e obter o maior número. A tela do jogo consiste em uma tabela, onde são possíveis os movimentos para cima, baixo, esquerda e direita. A cada movimento surge um novo número, que pode ser 2 ou 4, em um espaço vazio aleatório. A Figura 1 mostra uma representação do jogo.

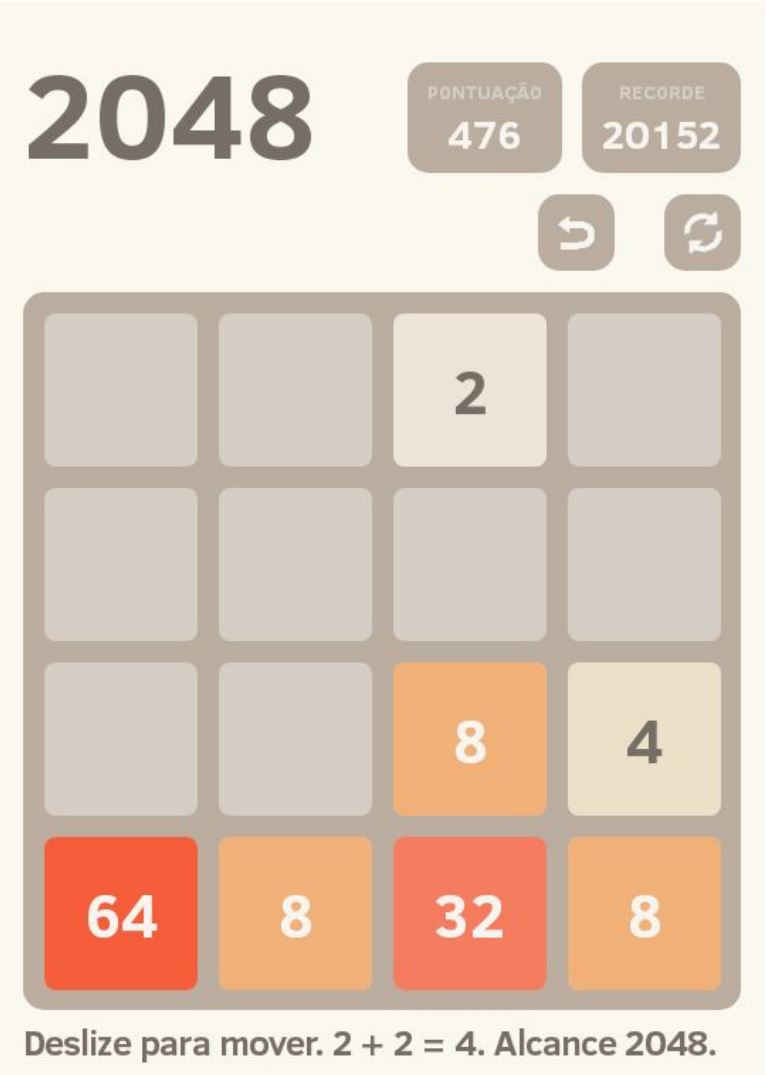


Figura 1 - Jogo 2048

CROMOSSOMOS E GENES

Para este problema foi escolhida a uma modelagem onde cada gene representa um movimento e um cromossomo representa uma sequência de movimentos. Os movimentos são representados em inteiros de 0 a 3, sendo esquerda, cima, direita e baixo respectivamente.

FITNESS

Para representar o fitness foi utilizado o score do jogo, ou seja, a melhor pontuação. A pontuação é uma soma de todas as junções no jogo, por exemplo: juntar 4+4 em um movimento acrescentará 8 no score.

OPERADORES GENÉTICOS

Neste trabalho foram aplicados os operadores de seleção, reprodução e mutação. Para a seleção foi utilizado o método roleta viciada, onde os cromossomos são dispostos em uma roleta tendo maior porcentagem dela de acordo com o fitness. Na reprodução foi utilizado o método de crossover com dois cortes, onde são escolhidos aleatoriamente dois pontos de corte para mesclar os genes dos pais. A mutação foi utilizada com uma porcentagem de 0,5%.

TESTES

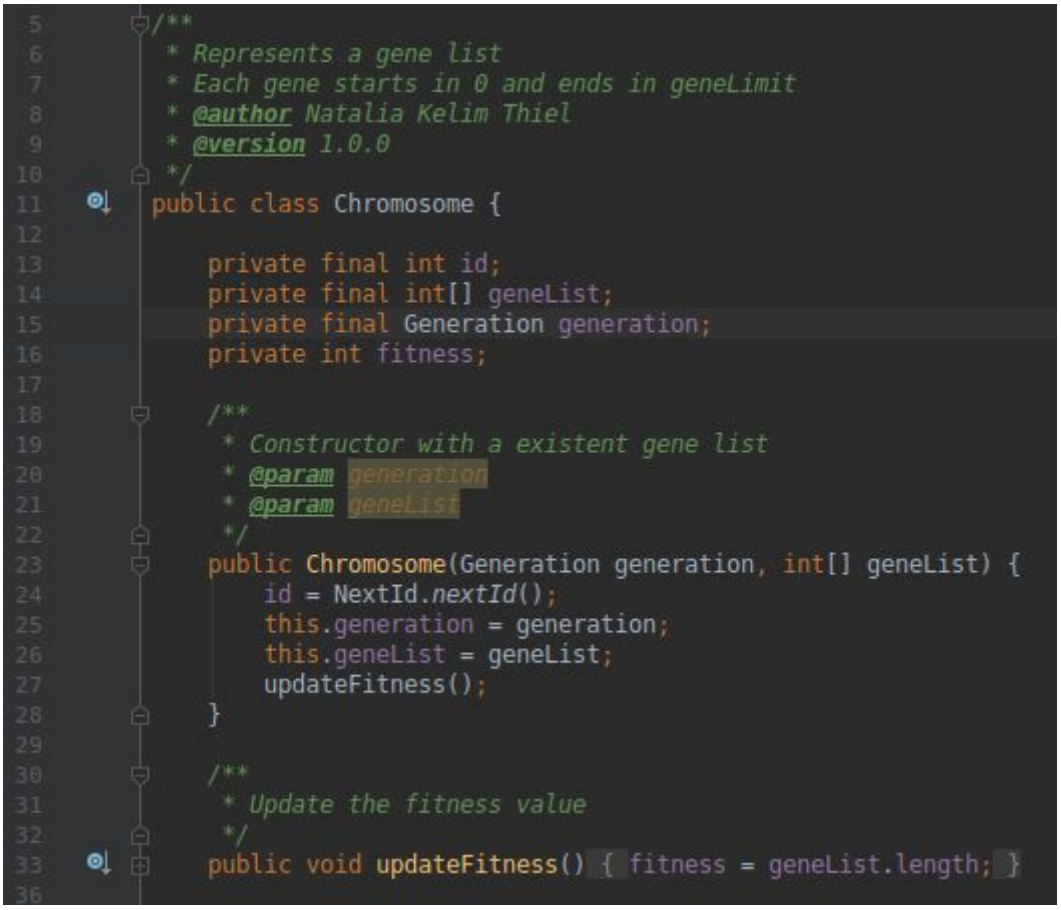
Os testes a seguir foram aplicados a fim de mostrar os resultados acerca do algoritmo genético criado para resolução do jogo 2048.

População	Movimentos	Mutação	Gerações	Fitness	Número
20	15	0,5%	10	112	32
20	20	0,5%	10	292	64
20	50	0,5%	10	464	64
4	50	0,5%	10	436	64
20	100	0,5%	10	1160	128

20	300	0,5%	10	2548	256
20	500	0,5%	10	2496	256
20	1000	0,5%	10	2250	256
50	300	0,5%	10	2956	256
50	300	1%	10	3016	256
50	300	2%	10	2544	256
20	300	0,5%	20	2768	256
15	300	0,5%	100	3428	256
26	300	0,5%	100	4164	512

CÓDIGO DOCUMENTADO

O código, desenvolvido em Java, foi separado em duas partes, a primeira é a resolução de algoritmos genéticos para alfabetos inteiros genéricos e depois é implementada no jogo 2048. A classe *Chromosome*, mostrada na Figura 2, representa um cromossomo, contendo um identificador único, uma lista de genes inteira, a geração onde ela pertence e seu fitness.



```java
/**
 * Represents a gene list
 * Each gene starts in 0 and ends in geneLimit
 * @author Natalia Kelim Thiel
 * @version 1.0.0
 */
public class Chromosome {

    private final int id;
    private final int[] geneList;
    private final Generation generation;
    private int fitness;

    /**
     * Constructor with a existent gene list
     * @param generation
     * @param geneList
     */
    public Chromosome(Generation generation, int[] geneList) {
        id = NextId.nextId();
        this.generation = generation;
        this.geneList = geneList;
        updateFitness();
    }

    /**
     * Update the fitness value
     */
    public void updateFitness() { fitness = geneList.length; }
```

Figura 2 - Classe Chromosome

O método *updateFitness* da linha 33 deve ser sobrescrito de acordo com a necessidade. A classe *Chromosome* também contém o método *mutation* que realiza a mutação em cada gene do cromossomo, mostrado na Figura 3. Na linha 42 são percorridos todos os genes e verificados se devem ser mutados de acordo com sua chance (linhas 43 e 44). A mutação ocorre na linha 50.

```java
    /**
     * Do the mutation
     */
    public void mutation() {
        ThreadLocalRandom r = ThreadLocalRandom.current();
        for (int i = 0; i < geneList.length; i++) {
            double chance = r.nextDouble();
            if (chance >= generation.getMutation()) {
                continue;
            }

            int mut;
            do {
                mut = r.nextInt(generation.getGeneLimit());
            } while (mut == geneList[i]);
            geneList[i] = mut;
        }
    }
```

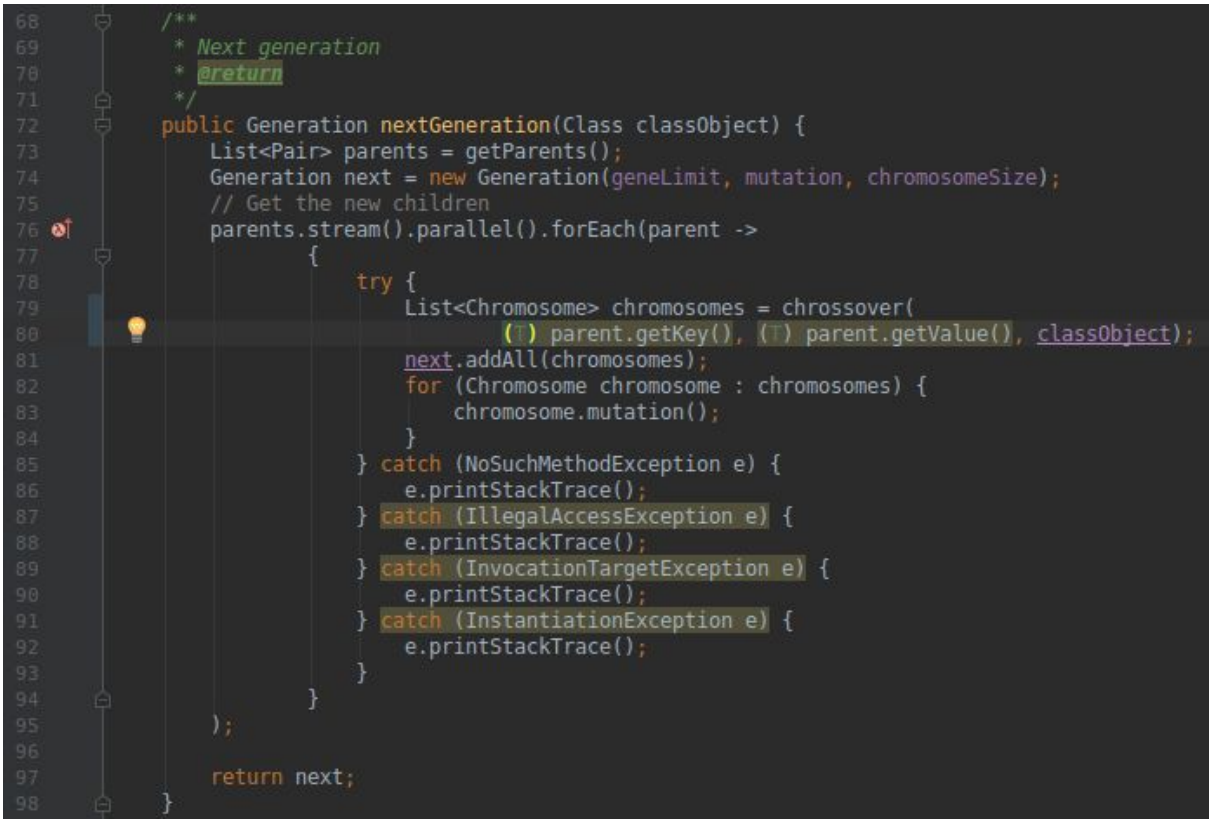
Figura 3 - Mutação

As gerações são representadas na classe *Generation*, Figura 4, que contém um id único, o limite do alfabeto (*geneLimit*), a taxa de mutação (*mutation*), o tamanho do cromossomo (*chromosomeSize*), a lista de cromossomos (*chromosomeList*) e a soma de todos os fitness (*totalFitness*). Esta classe também utiliza da característica *generics* do java, onde *T* pode ser qualquer classe que extenda *Chromosome*.

```
13  /**
14   * Represents the individuals of a generation
15   * @author Natalia Kelim Thiel
16   * @version 1.0.0
17   */
18  public class Generation <T extends Chromosome> {
19
20      private final int id;
21      private final int geneLimit;
22      private final double mutation;
23      private final int chromosomeSize;
24      private List<T> chromosomeList;
25      private int totalFitness;
26
27      /**
28       * Constructor with a new empty chromosome list
29       * @param geneLimit
30       * @param mutation
31       */
32      public Generation(int geneLimit, double mutation, int chromosomeSize) {
33          assert geneLimit > 0;
34          assert mutation >= 0 && mutation <= 1;
35          assert chromosomeSize > 0;
36
37          id = NextId.nextId();
38          this.geneLimit = geneLimit;
39          this.mutation = mutation;
40          this.chromosomeSize = chromosomeSize;
41          chromosomeList = new ArrayList<>();
42          totalFitness = 0;
43      }
```

Figura 4 - Classe Generation

O método para criação de uma nova geração a partir da anterior é o *nextGeneration*, onde é realizada a busca por pares (linha 73), aplicada a reprodução (linha 79 e 80) e a aplicada a mutação (linha 83). A Figura 5 mostra este método. O parâmetro *Class* é utilizado para criar instâncias de uma classe específica, podendo ser a classe *Chromosome* ou uma extensão.



```
68      /**
69       * Next generation
70       * @return
71       */
72      public Generation nextGeneration(Class classObject) {
73          List<Pair> parents = getParents();
74          Generation next = new Generation(geneLimit, mutation, chromosomeSize);
75          // Get the new children
76          parents.stream().parallel().forEach(parent ->
77                  {
78                      try {
79                          List<Chromosome> chromosomes = chrossover(
80                                  (T) parent.getKey(), (T) parent.getValue(), classObject);
81                          next.addAll(chromosomes);
82                          for (Chromosome chromosome : chromosomes) {
83                              chromosome.mutation();
84                          }
85                      } catch (NoSuchMethodException e) {
86                          e.printStackTrace();
87                      } catch (IllegalAccessException e) {
88                          e.printStackTrace();
89                      } catch (InvocationTargetException e) {
90                          e.printStackTrace();
91                      } catch (InstantiationException e) {
92                          e.printStackTrace();
93                      }
94                  }
95          );
96
97          return next;
98      }
```

Figura 5 - Método nextGeneration

A seleção de pares é mostrada na Figura 6, onde são gerados pares (linha 153) com o método de roleta viciada até serem válidos, ou seja, não se repetirem.

```
145     /**
146      * Random parents to build a next generation
147      * @return
148      */
149     protected List<Pair> getParents() {
150         updateTotalFitness();
151         List<Pair> parents = new ArrayList<>();
152         do {
153             Pair newParent = new Pair(getRouletteRandom(), getRouletteRandom());
154             if (isNewParentValid(newParent, parents)) {
155                 parents.add(newParent);
156             }
157         } while (parents.size() * 2 < chromosomeList.size());
158
159         return parents;
160     }
161
162     /**
163      * Find a chromosome by roulette method
164      * @return
165      */
166     protected Chromosome getRouletteRandom() {
167         double p = ThreadLocalRandom.current().nextDouble();
168
169         Chromosome parent = chromosomeList.get(chromosomeList.size() - 1);
170         double total = 0;
171         for (int i = 0; i < chromosomeList.size(); i++) {
172             Chromosome c = chromosomeList.get(i);
173             double value = c.getFitness() / (double) totalFitness;
174             total += value;
175             if (total >= p) {
176                 parent = c;
177                 break;
178             }
179         }
180         return parent;
181     }
```

Figura 6 - Método getParents e getRouletteRandom

A Figura 7 representa o operador de reprodução, crossover de 2 cortes. São gerados cortes aleatórios (linha 113), ordenados (linha 115) e aplicados (linhas 127 a 133).

```java
    /**
     * Make two children with the parents feature
     * @param mom
     * @param dad
     * @return
     */
    protected List<Chromosome> chrossover(Chromosome mom, Chromosome dad, Class classObject)
        ThreadLocalRandom r = ThreadLocalRandom.current();
        int[] children1 = new int[chromosomeSize];
        int[] children2 = new int[chromosomeSize];
        int nCuts = 2;
        int[] cut = new int[nCuts];
        for (int i = 0; i < nCuts; i++) {
            cut[i] = r.nextInt( origin: 1,  bound: mom.size() - 2);
        }
        Arrays.sort(cut);

        for (int i = -1; i < nCuts; i++) {
            int initial = 0;
            int limit = mom.size();
            if (i != -1) {
                initial = cut[i];
            }
            if (i != nCuts - 1) {
                limit = cut[i + 1];
            }
            for (int c = initial; c < limit; c++) {
                if (i % 2 == 0) {
                    children1[c] = mom.get(c);
                    children2[c] = dad.get(c);
                } else {
                    children2[c] = mom.get(c);
                    children1[c] = dad.get(c);
                }
            }
        }
```

Figura 7 - Método chrossover

Para testar e exemplificar este conjunto de classes, a Figura 8 mostra a programação.

```java
class GenerationTest {

    private static final int GENERATION_LENGTH = 10;
    private static final int CHROMOSOME_LENGTH = 6;
    private static final int GENE_LIMIT = 4;
    private static final double GENE_MUTATION = 0.005;
    private static Generation generation;

    @BeforeAll
    static void setUp() throws NoSuchMethodException, InstantiationException,
            IllegalAccessException, InvocationTargetException {
        generation = new Generation(GENE_LIMIT, GENE_MUTATION,
                CHROMOSOME_LENGTH, GENERATION_LENGTH, Chromosome.class);
        System.out.println(generation);
    }

    @Test
    void nextGeneration() {
        Generation next = generation.nextGeneration(Chromosome.class);
        System.out.println(next);
    }
```

Figura 8 - Testes de Geração

O resultado do teste é mostrado na Figura 9.

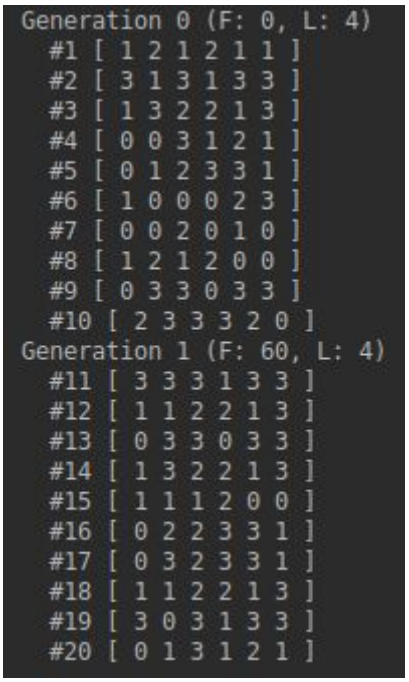
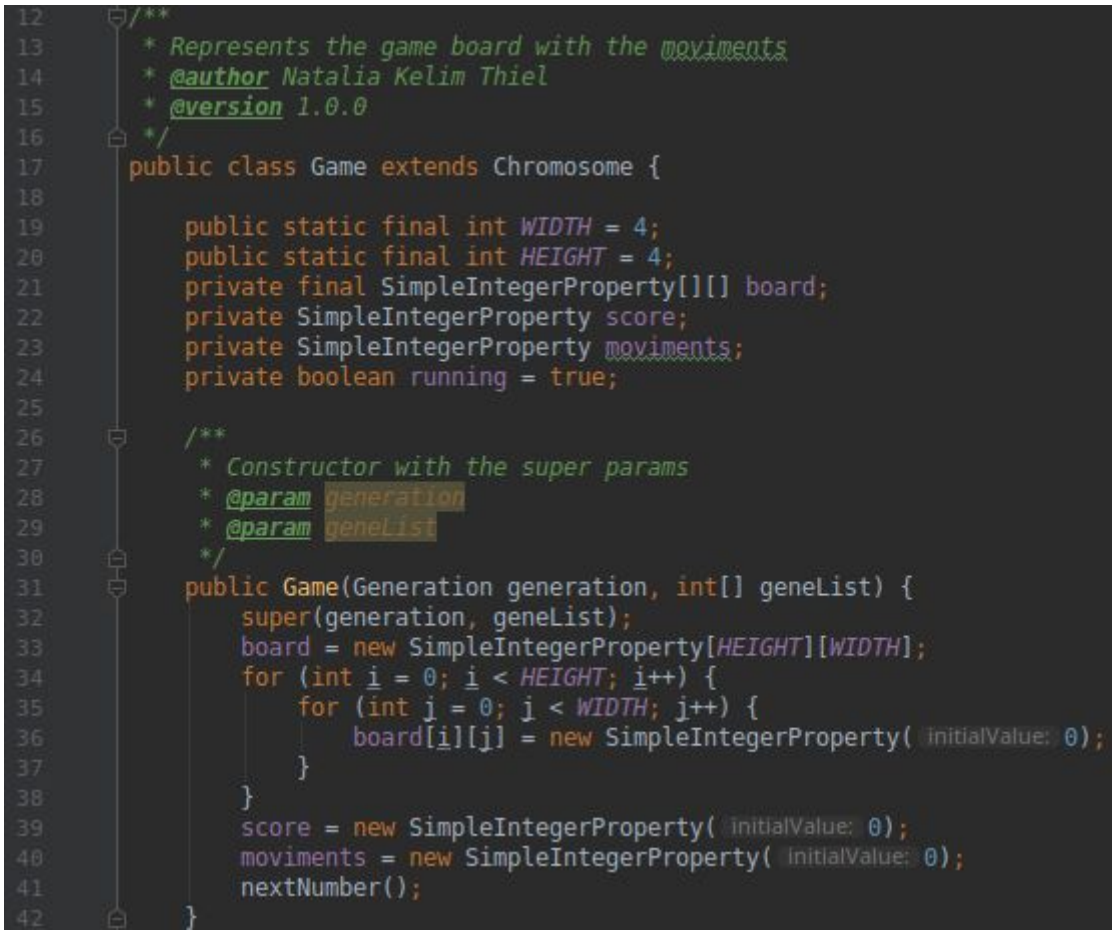


Figura 9 - Resultado dos Testes de Geração

A classe Game implementa a lógica do jogo 2048 e também estende a classe *Chromosome*, mostrada na Figura 10. Nela são armazenados o array do jogo, o score e o número de movimentos executados.



```java
/**
 * Represents the game board with the moviments
 * @author Natalia Kelim Thiel
 * @version 1.0.0
 */
public class Game extends Chromosome {

    public static final int WIDTH = 4;
    public static final int HEIGHT = 4;
    private final SimpleIntegerProperty[][] board;
    private SimpleIntegerProperty score;
    private SimpleIntegerProperty moviments;
    private boolean running = true;

    /**
     * Constructor with the super params
     * @param generation
     * @param geneList
     */
    public Game(Generation generation, int[] geneList) {
        super(generation, geneList);
        board = new SimpleIntegerProperty[HEIGHT][WIDTH];
        for (int i = 0; i < HEIGHT; i++) {
            for (int j = 0; j < WIDTH; j++) {
                board[i][j] = new SimpleIntegerProperty( initialValue: 0);
            }
        }
        score = new SimpleIntegerProperty( initialValue: 0);
        moviments = new SimpleIntegerProperty( initialValue: 0);
        nextNumber();
    }
```

Figura 10 - Classe Game

Para interface gráfica foi utilizado JavaFX. A Figura 11 mostra o resultado final de uma geração.

Figura 11 - Interface Gráfica

CONCLUSÃO

Mesmo com poucas gerações é possível visualizar uma conversão grande, entretanto não foi possível obter o valor 2048 em todos os testes, sendo o maior valor obtido em número 512 e score 4164. A implementação dos algoritmos genéticos se mostrou simples e seus resultados satisfatórios, sendo que 300 movimentos são considerados o ideal pois os valores acima não mostram mudança. Entretanto para uma melhor aplicação, como trabalhos futuros, implementar mais métodos de seleção e reprodução, a fim de comparar os resultados.