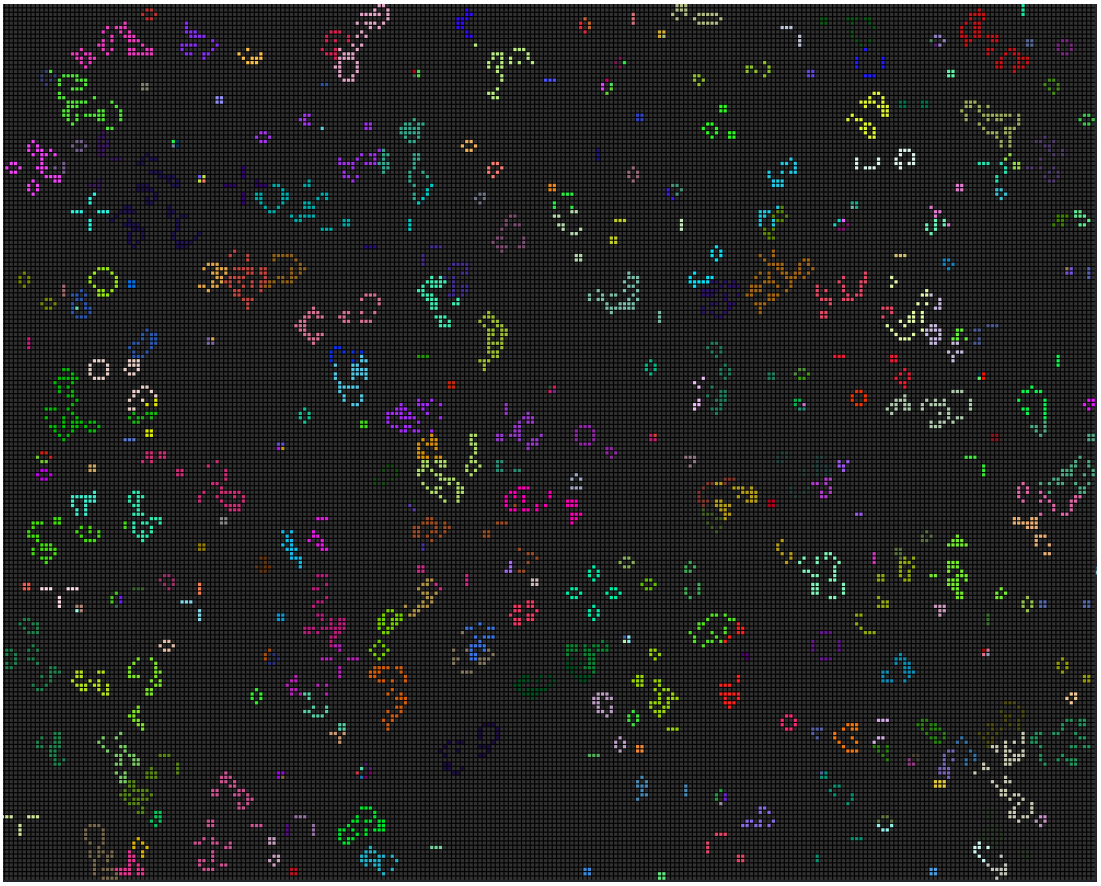
Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e Multimédia

Modelação e Simulação de Sistemas Naturais - MSSN - 2223SI

Projeto 1



Docente Paulo Vieira

Realizado por :
Pedro Silva 48965

23 de outubro de 2022

Conteúdo

1 Introdução **I**

2 DLA **I**
2.1 UML do DLA II
2.2 Reposição de partículas DLA II
2.3 Cor no DLA III

3 Jogo da Vida **IV**
3.1 Modo clássico V
3.2 Modo HighLife VI
3.3 Modo Immigration VI
3.4 Modo QuadLife VII
3.5 Modo Life-Colorido VII

4 Conclusões **VIII**

Lista de Figuras

1 DLA (antes) I
2 DLA (depois) I
3 UML do DLA II
4 Método "draw"modificado II
5 Método "updateState"modificado III
6 Jogo da Vida (clássico) IV
7 Jogo da Vida (colorido) IV
8 Método Life V
9 Método getAliveNeighbours V
10 Método HighLife VI
11 Immigration VI
12 QuadLife VII
13 Método Life_Color VII
14 Método getDominantColor VII

1 Introdução

Este projeto pretende consolidar e avaliar os conhecimentos adquiridos na primeira parte da disciplina Modelação e Simulação de Sistemas Naturais tendo como objetivo o desenvolvimento de um DLA (Agregação de Difusão Limitada) e do Jogo da Vida (versão 23/3). O DLA é um processo no qual partículas submetidas a passeio aleatório aglomeram-se para formar agregados de tais partículas. O jogo da vida é o exemplo mais bem conhecido de autómato celular. Foi criado de modo a reproduzir, através de regras simples, as alterações e mudanças em grupos de seres vivos, tendo aplicações em diversas áreas da ciência.

2 DLA

Um dos objetivos deste projeto foi, tendo como base as classes disponibilizadas, Walker e DLA, a conclusão da implementação do algoritmo de agregação por difusão, fazendo com que o número de partículas em movimento seja sempre constante ao longo do tempo. Também é necessário atribuir cor às partículas paradas e/ou em movimento.

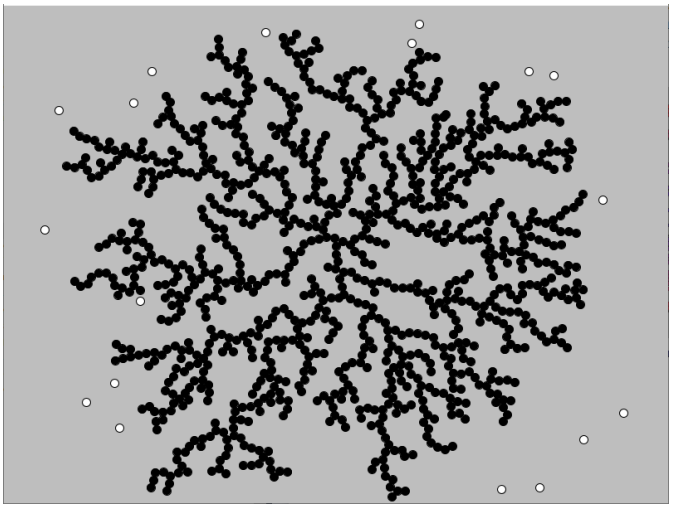


Figura 1: DLA (antes)

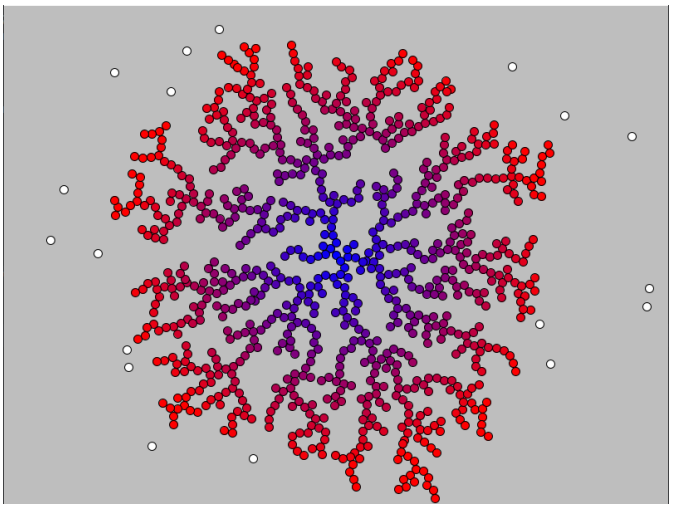


Figura 2: DLA (depois)

2.1 UML do DLA

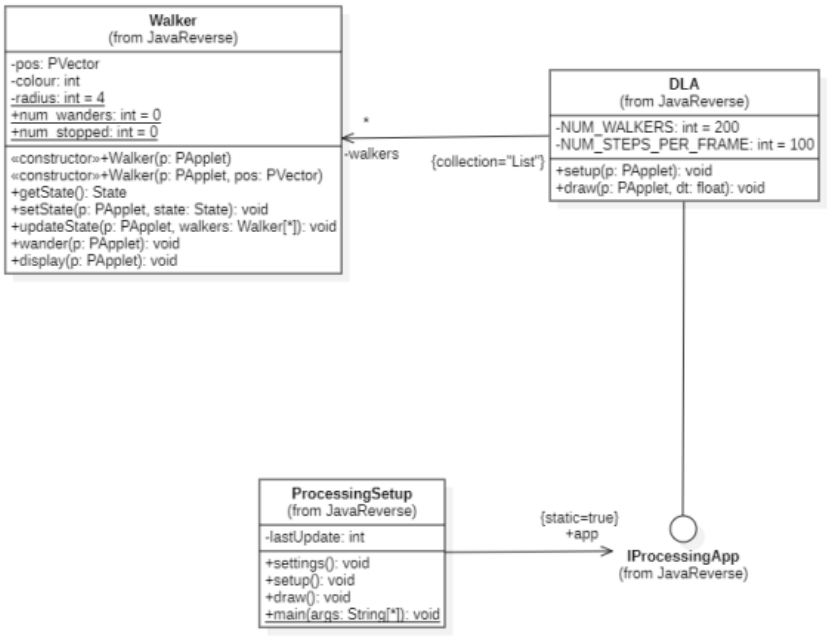


Figura 3: UML do DLA

2.2 Reposição de partículas DLA

```
@Override
public void draw(PApplet p, float dt)
{
    p.background( rgb: 190);

    for (int i = 0; i<num_Steps;i++){
        for (Walker w : walkers) {
            if (w.getState() == Walker.State.WANDER) {
                w.wander(p);
                w.updateState(p, walkers);
            }
        }
    }

    for (Walker w : walkers) {
        w.display(p);
    }

    while (Walker.num_wanders!=num_Walkers)
        walkers.add(new Walker(p));

    System.out.println("Stopped = " + Walker.num_stopped + "Wanders = " + Walker.num_wanders);
}
```

Figura 4: Método "draw"modificado

Para repor o número de partículas em movimento para que este seja sempre constante ao longo do tempo foi necessário adicionar ao método "draw"da classe "DLA"o conteúdo destacado na figura 4. Foi acrescentado um "while"que enquanto o número de "walkers"não for igual ao número inicial é adicionado um novo "walker".

2.3 Cor no DLA

O critério escolhido para a atribuição das cores das partículas foi a distancia da partícula parada à partícula do centro em que à medida que a estas se afastam do centro conseguimos observar um gradiente de azul para vermelho.

```
public void updateState(PApplet p,List<Walker> walkers)
{
    if (state == State.STOPPED)
        return;

    for (Walker w: walkers){
        if (w.state == State.STOPPED){
            float dist = PVector.dist(pos, w.pos);
            if (dist < 2*radius){
                setState(p, State.STOPPED);
                num_wanders--;
                break;
            }
            PVector center = new PVector( x: 400,  y: 300);
            float dist_center = PVector.dist(w.pos,center);
            int r=0;
            int g=0;
            int b=255;
            w.color=p.color(r+dist_center,g,b-dist_center);
        }
    }
}
```

Figura 5: Método "updateState"modificado

Para alterar a cor das partículas paradas foi necessário adicionar ao método "updateState"da classe "Walker"o conteúdo destacado na figura 3. Foi criado uma variável "PVector center"onde é guardado as coordenadas do centro e uma variável "float dist_center"onde vai ser guardada a distância da partícula que acabou de parar ao centro utilizando o método "PVector.dist()". Depois mudamos a cor da partícula ao retirar do valor de azuis o valor da distancia e adicionar ao valor dos vermelhos.

3 Jogo da Vida

O outro objetivo deste projeto foi, tendo como base as classes disponibilizadas, CellularAutomata e Cell, a implementação da versão clássica do Jogo da Vida. As regras definidas são aplicadas a cada nova "geração", assim, a partir de uma imagem num tabuleiro bi-dimensional definida pelo jogador, percebem-se mudanças muitas vezes inesperadas a cada nova geração, variando de padrões fixos a caóticos. Foi requisitado de modo facultativo a implementação de variantes do Jogo da Vida e a atribuição de cores às células de acordo com um dado critério.

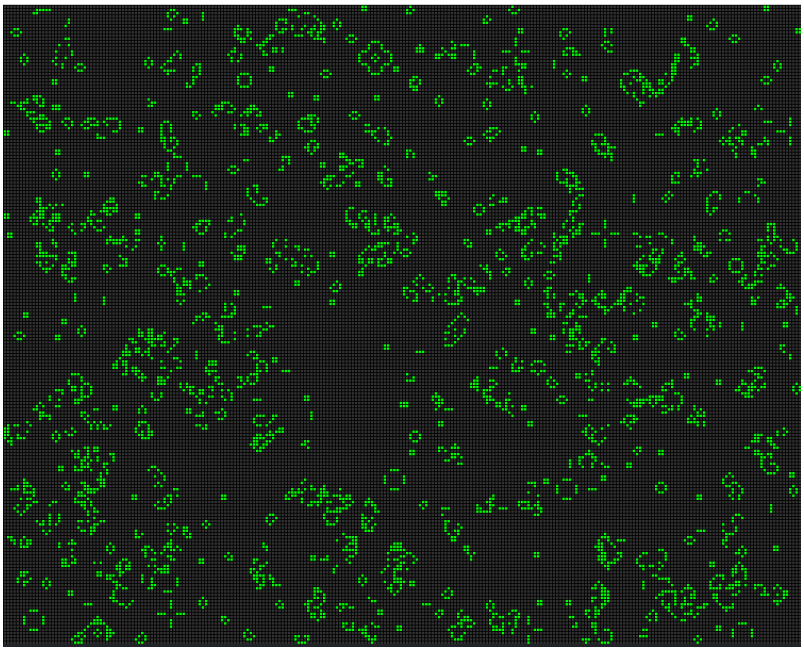


Figura 6: Jogo da Vida (clássico)

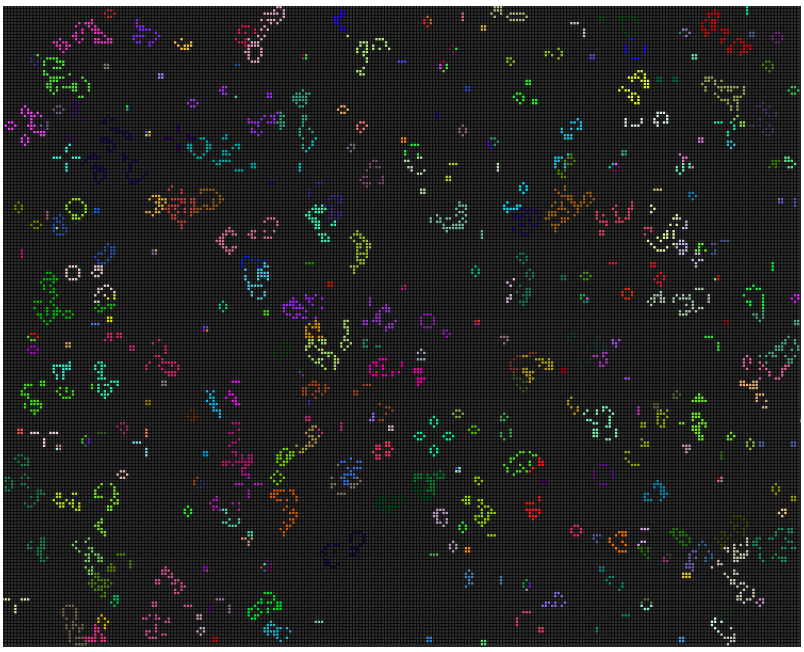


Figura 7: Jogo da Vida (colorido)

3.1 Modo clássico

O jogo da vida passa-se num arranjo bidimensional de células que podem estar em um de dois estados, vivo ou morto. Cada célula interage com as suas oito vizinhas, as células adjacentes horizontal, vertical e diagonalmente. O jogo evolui em unidades de tempo chamadas gerações. A cada nova geração, o estado do jogo é atualizado pela aplicação das seguintes regras:

- Uma célula morta com exatamente três vizinhos vivos torna-se viva (nascimento).
- Uma célula viva com menos de dois vizinhos vivos morre por isolamento.
- Uma célula viva com mais de três vizinhos vivos morre por superpopulação.
- Uma célula viva com dois ou três vizinhos vivos permanece viva.

```
public void Life() {

    int[][] cellsBuffer = new int[nRows][nCols];

    for (int x = 0; x<nRows; x++) {
        for (int y=0; y<nCols; y++) {
            cellsBuffer[x][y] = cells[x][y].getState();
        }
    }

    for (int i = 0; i < nRows; i++) {
        for (int j = 0; j < nCols; j++) {
            int neighbours = getAliveNeighbours(cellsBuffer,i,j);
            if (cellsBuffer[i][j]==1) {
                if (neighbours < 2 || neighbours > 3)
                    cells[i][j].setState(0);
            }
            else {
                if (neighbours == 3 )
                    cells[i][j].setState(1);
            }
        }
    }
}
```

Figura 8: Método Life

```
public int getAliveNeighbours(int[][] cellsBuffer,int i, int j )
{
    int neighbours = 0;
    for (int ii=i-1; ii<=i+1;ii++) {
        for (int jj=j-1; jj<=j+1;jj++) {
            if (((ii>=0)&&(ii<nRows))&&((jj>=0)&&(jj<nCols))) {
                if (!((ii==i)&&(jj==j))) {
                    if (cellsBuffer[ii][jj]==1)
                        neighbours ++;
                }
            }
        }
    }
    return neighbours;
}
```

Figura 9: Método getAliveNeighbours

Para traduzir estas regras em código temos o método "Life"(figura 8), que começa por "tirar uma foto"do tabuleiro ao guardar os estados de cada célula num array bidimensional. Depois percorremos os vizinhos de cada célula e obtemos o número de vizinhos vivos da mesma, ao chamar o método "getAliveNeighbours"que ao receber a célula percorre os seus vizinhos tendo parâmetros para avaliar se os vizinhos estão fora do tabuleiro. Tendo o número de vizinhos vivos, se a célula estiver viva e tiver menos de 2 ou mais de 3 vizinhos vivos morre. Se estiver morta e tiver 3 vizinhos vivos esta fica viva. Numa fase inicial em vez de um array bidimensional tentámos utilizar uma nova instância da classe "Cell"mas observámos que cada vez que mudávamos a célula auxiliar de estado esta também alterava a o estado da célula principal o que não era o desejado.

3.2 Modo HighLife

O modo HighLife é idêntico ao modo clássico, a única diferença é na regra de nascimento que em vez de uma célula morta renascer apenas com 3 vizinhos vivos isto também acontece com 6 vizinhos. Podemos observar esta mudança comparando o código destacado na figura 10 com o mesmo código na figura 8.

```
public void HighLife() {

    int[][] cellsBuffer = new int[nRows][nCols];

    for (int x = 0; x<nRows; x++) {
        for (int y=0; y<nCols; y++) {
            cellsBuffer[x][y] = cells[x][y].getState();
        }
    }
    for (int i = 0; i < nRows; i++) {
        for (int j = 0; j < nCols; j++) {
            int neighbours = getAliveNeighbours(cellsBuffer,i,j);
            if (cellsBuffer[i][j]==1) {
                if (neighbours < 2 || neighbours > 3)
                    cells[i][j].setState(0);
            }
            else {
                if (neighbours == 3 || neighbours==6 )
                    cells[i][j].setState(1);
            }
        }
    }
}
```

Figura 10: Método HighLife

3.3 Modo Immigration

As regras do modo Immigration são iguais às do modo clássico, mas em vez de termos apenas um estado vivo temos dois, representados por vermelho e azul. Quando uma célula nasce esta fica com a cor da maior parte dos vizinhos, ou seja, se esta tem 2 ou 3 vizinhos vermelhos nasce com a cor vermelha e vice-versa. Neste modo podemos observar comportamentos interessantes como osciladores com maior período ou gliders de duas cores.

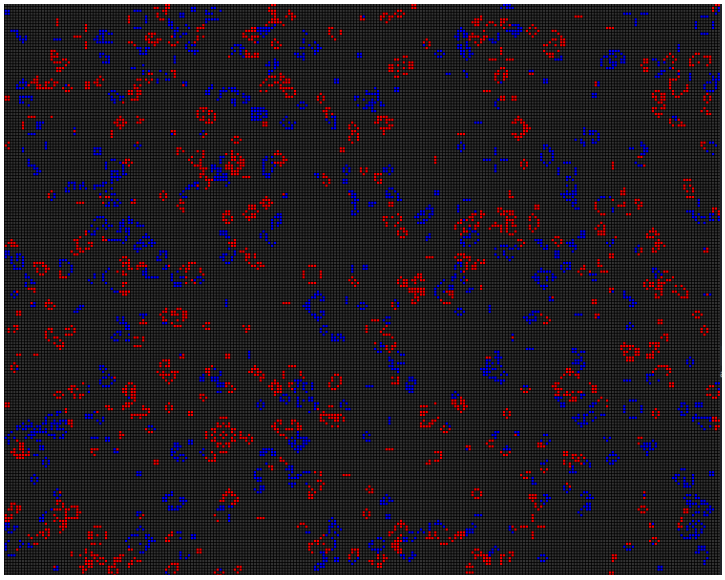


Figura 11: Immigration

3.4 Modo QuadLife

O modo QuadLife tem o mesmo conceito do modo Immigration, mas em vez de 2 estados vivos temos quatro. Quando uma célula nasce rodeada de três estados vivos diferentes esta nasce com a cor do estado não presente, se isto não acontecer a célula nasce de acordo com a regra da maioria assim como no modo anterior.

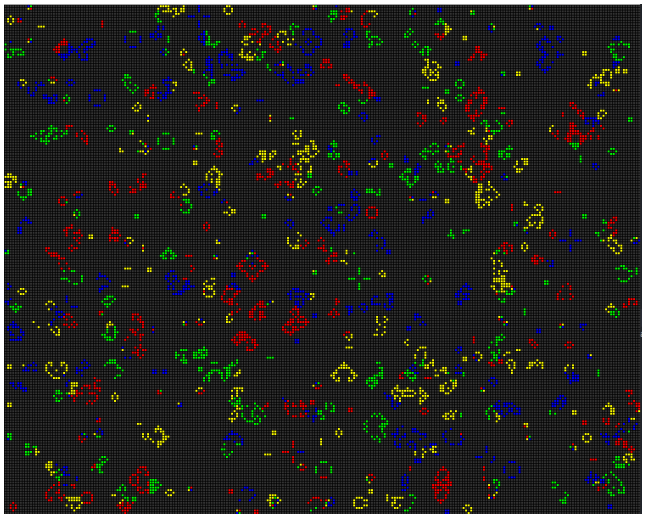


Figura 12: QuadLife

3.5 Modo Life-Colorido

Este modo é uma combinação de todos os modos anteriores possuindo as regras do modo clássico acrescentando a variante de todas as células vivas terem uma cor qualquer na primeira geração. O método "Life_Color"(figura 13) funciona de maneira semelhante ao do método "Life"(figura 8) sendo a única diferença a atribuição de uma cor à célula que nasce. Conseguimos fazê-lo a partir do método "getDominantColor"que devolve a cor dominante entre os vizinhos se existir um empate entre as cores, ou seja, os vizinhos têm todos cores diferentes devolve uma dessas cores aleatoriamente.

```
public void Life_Color() {

    int[][] cellsBuffer = new int[nRows][nCols];

    for (int x = 0; x<nRows; x++) {
        for (int y=0; y<nCols; y++) {
            cellsBuffer[x][y] = cells[x][y].getState();
        }
    }

    for (int i = 0; i < nRows; i++) {
        for (int j = 0; j < nCols; j++) {
            int neighbours = getAliveNeighbours(cellsBuffer,i,j);
            if (cellsBuffer[i][j]==1) {
                if (neighbours < 2 || neighbours > 3)
                    cells[i][j].setState(0);
            }
            else {
                if (neighbours == 3 ) {
                    cells[i][j].setState(1);
                    cells[i][j].setColor(getDominantColor(cellsBuffer,i,j));
                }
            }
        }
    }
}
```

Figura 13: Método Life_Color

```
public int getDominantColor(int[][] cellsBuffer,int i, int j )
{
    int counter=0;
    int[] colors = new int[3];

    for (int ii=i-1; ii<=i+1;ii++) {
        for (int jj=j-1; jj<=j+1;jj++) {
            if (((ii>=0)&&(ii<nRows))&&((jj>=0)&&(jj<nCols))) {
                if (!((ii==i)&&(jj==j))) {
                    if (cellsBuffer[ii][jj]==1) {
                        colors[counter] = cells[ii][jj].getColor();
                        counter++;
                    }
                }
            }
        }
    }
    Random randomNum = new Random();
    if (colors[0] == colors[1])
        return colors[0];
    if (colors[0] == colors[2])
        return colors[0];
    if (colors[1] == colors[2])
        return colors[1];
    else
        return colors[1 + randomNum.nextInt( bound: 2)];
}
```

Figura 14: Método getDominantColor

4 Conclusões

O objetivo deste projeto era o de conceber um DLA e um Jogo da Vida utilizando todos os conhecimentos adquiridos ao longo de Modelação e Simulação de Sistemas Naturais.Gostava de ter adicionado uma GUI que servisse de "launcher"para ambos os projetos e que pudesse alterar as suas configurações mas devido à falta de tempo isto não foi possível. Porém Creio que ambos o DLA e o Jogo da Vida assim como as suas partes facultativas estão de acordo com o pretendido pelos docentes, demonstrando o domínio que tenho sobre a matéria lecionada.