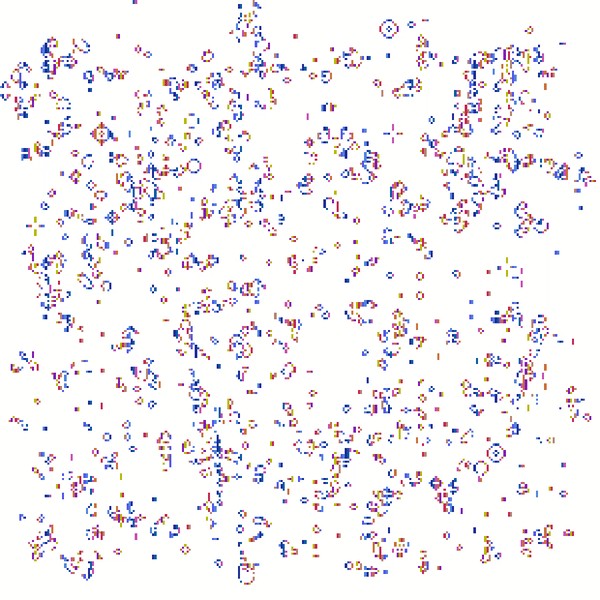
Modelação e Simulação de

Sistemas Naturais

20/21 – Semestre Inverno

Trabalho nº1



LEIM33D-A – 8 novembro 2020

Luís Fonseca nº 45125

Paulo Jorge nº45121

Índice

[Introdução 2](#_Toc55136881)

[Jogo Da Vida 3](#_Toc55136882)

[Conclusões 5](#_Toc55136883)

[Bibliografia 6](#_Toc55136884)

[Anexos 7](#_Toc55136885)

Índice de Figuras

[Figura 1 - Resultado obtido quando é corrido o jogo da vida 4](#_Toc55136888)

# Introdução

Neste trabalho foi feito o estudo do comportamento de agentes autónomos e DLA(Diffusion-Limited Aggregation). Com estes conceitos, foi proposto um trabalho de casa onde era posto em prática, a execução destes conceitos, pegando em vários exercícios e gerar diferentes resultados para eventuais estudos e conclusões.

# Jogo Da Vida

Neste exercício, era pedido que fosse feita uma implementação do jogo da vida, em java, usando agentes autónomos. Para isso foram construídas várias classes para a concretização deste exercício. Todo o código feito para este exercício pode ser encontrado na secção “Anexos”.

Como ponto de partida, o grupo criou a classe *Cell*, classe esta que corresponde ao desenho de uma célula, neste caso são inicializadas todas as variáveis, e feitos os *get* e *set.*

Foi criada também a classe *CellularAutomata*, que permite aplicar diferentes comportamentos às células, nomeadamente, criar células vizinhas à sua volta, construir uma grelha (que irá ser 2D) e mostrar o que se pretende desta classe(usando o método *display())*. É nesta classe que se constrói a grelha2D, dando o nome a esse método de *createGrid()*.

De seguida, foram criados os métodos *setNeighbors()* e *setNeighbors4()*, ou seja, permite criar vizinhos à volta de cada célula

No final foi criado o método *setRandomState()* que permite fazer uma escolha aleatória do estado, que pode ser 0, caso essa célula esteja morta, ou o valor de 1, caso esteja viva.

Foi também criada a classe *LifeCell* que corresponde ao tempo de vida de uma célula, para esta classe foram criados os métodos: *flipState()* que corresponde em alterar o estado, de 0 para 1 e vice-versa, o método *countAlives()* que permite contar o número de células vivas, e o método *applyRule()* que permite aplicar uma determinada regra.

Como última classe, foi criada a classe *JogoDaVida* que permite correr o nosso jogo da vida, para isso foi criado o método *createGrid()* que permite criar uma grelha, e colocar a respetiva célula na grelha, e colocar os vizinhos mais próximos dessa célula, evocando o método *setNeighbors()*. No final o método *run()* que permite correr a classe criada, evocando os métodos *countAlives()* e *applyRule()*.

No final, foi criada a classe *JogoDaVidaApp,* que serve apenas para fazer correr o jogo da vida. A execução do programa pode ser visualizada na imagem em baixo:

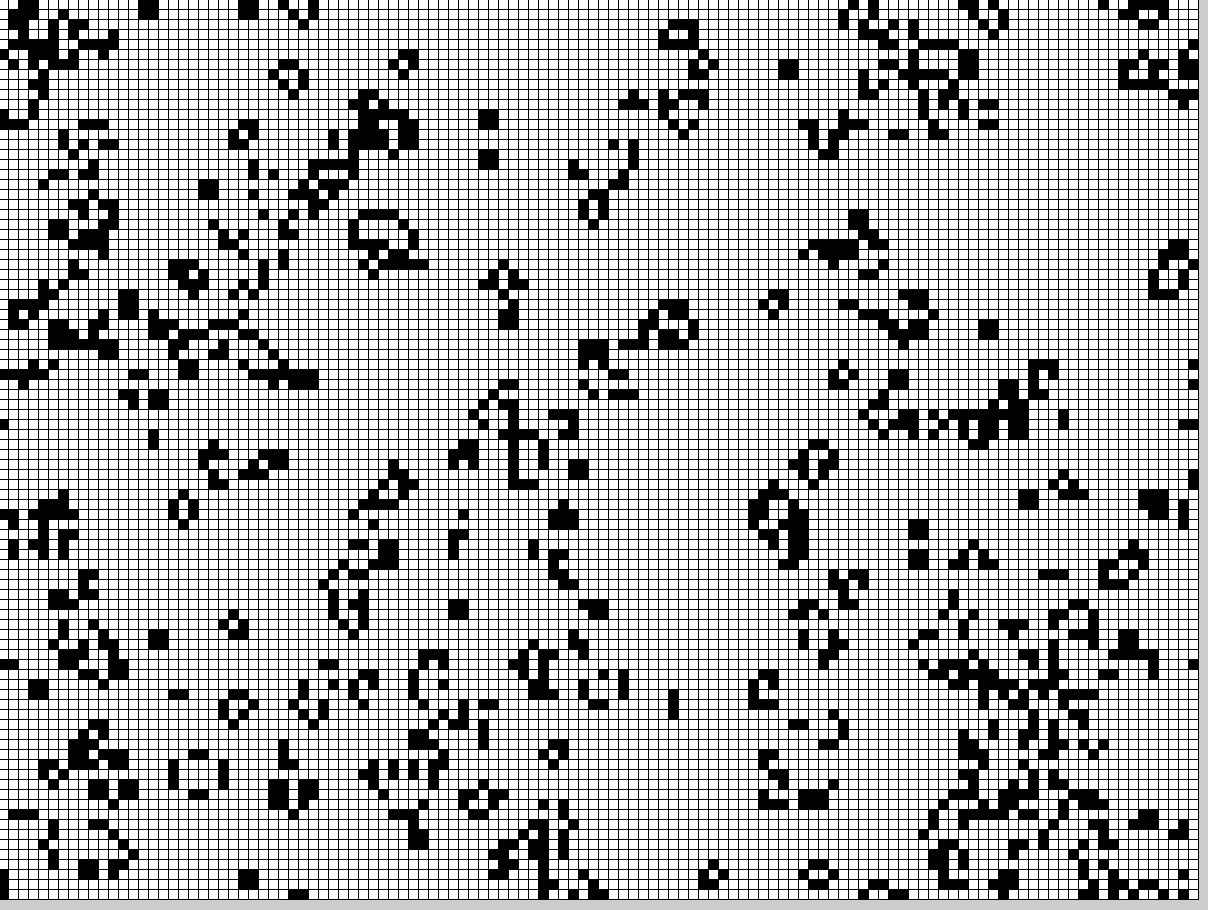


Figura 1 - Resultado obtido quando é corrido o jogo da vida

# DLA

Agregação por difusão limitada (DLA, do [inglês](https://pt.wikipedia.org/wiki/L%C3%ADngua_inglesa) Diffusion-limited aggregation) é um processo no qual partículas que são submetidas a um [passeio aleatório](https://pt.wikipedia.org/wiki/Passeio_aleat%C3%B3rio) se aglomeram para formar agregados. Esta teoria, proposta por Witten e Sander em 1981, é aplicável a agregação de qualquer sistema onde a [difusão](https://pt.wikipedia.org/wiki/Difus%C3%A3o) é o meio primário de transporte no sistema. DLA pode ser observado em muitos sistemas tais como eletrodeposição, fluxo de Hele-Shaw, depósitos minerais, e rotura de dielétrico.

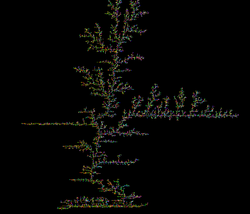
[](https://pt.wikipedia.org/wiki/Ficheiro:Brownian_tree_vertical_large.png)

Figura - Uma árvore browniana de uma simulação computacional

As agregações formadas em processos DLA são referenciadas como [árvores brownianas](https://pt.wikipedia.org/w/index.php?title=%C3%81rvore_browniana&action=edit&redlink=1). Esses agrupamentos são um exemplo de um [fractal](https://pt.wikipedia.org/wiki/Fractal). A duas dimensões, esses fractais exibem uma dimensão de aproximadamente 1,71 por partícula livre que são irrestritas por uma fronteira, entretanto numa simulação computacional numa região restrita irá alterar a [dimensão do fractal](https://pt.wikipedia.org/wiki/Dimens%C3%A3o_fractal) levemente para um DLA na mesma [dimensão encaixante](https://pt.wikipedia.org/w/index.php?title=Dimens%C3%A3o_encaixante&action=edit&redlink=1). Algumas variações também são observadas, dependendo da geometria do crescimento, quer seja de um único ponto radialmente para fora ou de um plano ou linha por exemplo.

# Conclusões

Neste trabalho de casa, o grupo conseguiu aprender mais acerca de fractais e agentes autónomos. Através da execução dos diferentes exercícios proposto do trabalho de casa, o grupo conseguiu fazer quase todos os exercícios, não sendo possível a execução de todos, surgindo algumas dúvidas e complicações no meio. Apesar de algumas dificuldades sentidas, o grupo conseguiu ficar a aprender estes conceitos e consolidar esta matéria, indo com uma “bagagem” que podem vir a ser úteis para futuros trabalhos de casa e para o projeto final da disciplina.

# Bibliografia

Folhas fornecidos pelo docente Arnaldo Abrantes

DLA

* <https://pt.wikipedia.org/wiki/Agrega%C3%A7%C3%A3o_por_difus%C3%A3o_limitada>

Autómatos Celulares:

* <https://en.wikipedia.org/wiki/Cellular_automaton>
* <http://mathworld.wolfram.com/ElementaryCellularAutomaton.html>
* <https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life>

# Anexos

* **Classe Cell**

**public** **class** Cell {

**protected** CellularAutomata ca;

**protected** **int** row, col;

**protected** **int** state;

**protected** Cell[] neighbors;

**protected** **int** w,h;

**public** Cell(CellularAutomata ca, **int** row, **int** col) {

**this**.ca = ca;

**this**.row = row;

**this**.col = col;

state = 0;

neighbors = **null**;

w = ca.getCellWidth();

h = ca.getCellHeight();

}

**public** **void** setNeighbors(Cell[] neighbors) {

**this**.neighbors = neighbors;

}

**public** Cell[] getNeighbors() {

**return** neighbors;

}

**public** **void** setState(**int** state) {

**this**.state = state;

}

**public** **int** getState() {

**return** state;

}

**public** PVector getCenter() {

**float** x = (col + 0.5f)\*w;

**float** y = (row + 0.5f)\*h;

**return** **new** PVector (x,y);

}

**public** **void** display(PApplet p) {

p.pushStyle();

p.fill(ca.getStateColors()[state]);

p.rect(col\*w, row\*h, w, h);

p.popStyle();

}

}

* **Classe CellularAutomata**

**public** **class** CellularAutomata {

**protected** **int** nrows,ncols;

**protected** **int** w,h;

**protected** Cell[][] cells;

**protected** **int** radius;

**protected** **boolean** moore;

**protected** **int** numberofStates;

**protected** **int**[] colors;

**protected** PApplet p;

**public** CellularAutomata(PApplet p, **int** nrows, **int** ncols, **int** radius, **boolean** moore, **int**  numberofStates) {

**this**.p = p;

**this**.nrows = nrows;

**this**.ncols = ncols;

**this**.radius = radius;

**this**.moore = moore;

**this**.numberofStates = numberofStates;

w = p.width/ncols;

h = p.height/nrows;

cells = **new** Cell[nrows][ncols];

createGrid();

colors = **new** **int**[numberofStates];

}

**protected** **void** createGrid() {

**for**(**int** i = 0; i < nrows; i++) {

**for**(**int** j = 0; j <ncols ; j++) {

cells[i][j] = **new** Cell(**this**, i ,j);

}

}

**if**(moore) setNeighbors();

**else** setNeighbors4();

}

**public** Cell getCellGrid(**int** row,**int** col) {

**return** cells[row][col];

}

**protected** **void** setNeighbors4() {

**int** numberofNeighbors = 2\*(radius^2+radius)+1;

**for**(**int** i = 0; i < nrows; i++) {

**for**(**int** j = 0; j <ncols ; j++) {

**int** n = 0;

Cell[] neigh = **new** Cell[numberofNeighbors];

**for**(**int** ii = -radius; ii <= radius ; ii++) {

**for**(**int** jj =-radius+Math.*abs*(ii) ; jj <= radius-Math.*abs*(ii); jj++) {

**int** row = (i + ii + nrows) % nrows;

**int** col = (j + jj + ncols) % ncols;

neigh[n++] = cells[row][col];

}

}

cells[i][j].setNeighbors(neigh);

}

}

}

**protected** **void** setNeighbors() {

**for**(**int** i = 0; i < nrows; i++) {

**for**(**int** j = 0; j <ncols ; j++) {

Cell[] neigh = **new** Cell[(**int**)Math.*pow*(2\*radius+1, 2)];

**int** n = 0;

**for**(**int** ii = -radius; ii <= radius ; ii++) {

**for**(**int** jj =-radius ; jj <= radius; jj++) {

**int** row = (i + ii + nrows) % nrows;

**int** col = (j + jj + ncols) % ncols;

neigh[n++] = cells[row][col];

}

}

cells[i][j].setNeighbors(neigh);

}

}

}

**public** **void** reset() {

**for**(**int** i = 0; i < nrows; i++) {

**for**(**int** j = 0; j < ncols ; j++) {

cells[i][j].setState(0);

}

}

}

**public** **int** getCellWidth() {

**return** w;

}

**public** **int** getCellHeight() {

**return** h;

}

**public** **void** setStateColors(**int**[] colors) {

**this**.colors = colors;

}

**public** **int**[] getStateColors() {

**return** **this**.colors;

}

**public** **int** getNumberOfStates() {

**return** numberofStates;

}

**public** Cell getCell(**int** x, **int** y) {

**int** row = y/h;

**int** col = x/w;

**if**(row >= nrows) row = nrows-1;

**if**(col >= ncols) col = col-1;

**return** cells[row][col];

}

**public** **void** setRandomState() {

**for**(**int** i = 0; i < nrows; i++) {

**for**(**int** j = 0; j < ncols ; j++) {

cells[i][j].setState((**int**) p.random(numberofStates));

}

}

}

**public** **void** setRandomStateCustom(**double**[] pmf) {

CustomRandomGenerator crg = **new** CustomRandomGenerator(pmf);

**for**(**int** i = 0; i < nrows; i++) {

**for**(**int** j = 0; j < ncols ; j++) {

cells[i][j].setState(crg.getRandomClass());

}

}

}

**public** **void** display() {

**for**(**int** i = 0; i < nrows; i++) {

**for**(**int** j = 0; j < ncols ; j++) {

cells[i][j].display(**this**.p);

}

}

}

* **Classe LifeCell**

**public** **class** LifeCell **extends** Cell{

**private** **int** nAlives;

**public** LifeCell(CellularAutomata ca, **int** row,**int** col) {

**super**(ca, row, col);

}

**public** **void** flipState() {

**if**(state == 0) state =1;

**else** state = 0;

}

**public** **void** countAlives() {

nAlives = 0;

**for**(Cell c: neighbors) nAlives += c.state;

nAlives -= state;

}

**public** **void** applyRule() {

**if**(state == 0 && nAlives == 3) state =1;

**if**(state ==1 && (nAlives <2 || nAlives >3)) state=0;

}

}

* **Classe JogoDaVida**

**public** **class** JogoDaVida **extends** CellularAutomata{

**public** JogoDaVida(PApplet p, **int** nrows, **int** ncols) {

**super**(p,nrows,ncols,1,**true**,2);

}

**public** **void** initRandom(**float** prob) {

**double**[] pmf = **new** **double**[2];

pmf[0] = 1-prob;

pmf[1] = prob;

setRandomStateCustom(pmf);

}

**protected** **void** createGrid() {

**for**(**int** i = 0; i<nrows;i++) {

**for**(**int** j = 0; j<ncols;j++) {

cells[i][j]=**new** LifeCell(**this**,i,j);

}

}

setNeighbors();

}

**public** **void** run() {

LifeCell c;

**for**(**int** i = 0; i<nrows;i++) {

**for**(**int** j = 0; j<ncols;j++) {

c = (LifeCell) cells[i][j];

c.countAlives();

}

}

**for**(**int** i = 0; i<nrows;i++) {

**for**(**int** j = 0; j<ncols;j++) {

c = (LifeCell) cells[i][j];

c.applyRule();

}

}

}

}