8. CA 伝染病モデル

2014/12/10

1　はじめに

2 次元のセルオートマトンの応用として、伝染病モデルを扱います。各セルには、一つの個体を置くことにします。空のセルも可能とします。システムは周期境界条件に従うとします。2 次元の周期境界ですから、左右に閉じていると同時に上下にも閉じています。ドーナツのような形ですが、数学ではトーラス (torus) と呼びます。

表 1: 伝染病モデルにおける状態

S	健康な状態。隣接する状態 I の個体から病気をうつされる。
I	病気の状態。隣接する状態 S の個体に病気をうつす。
R	治癒した状態。免疫を得て、病気に罹らない。

個体の状態は $s \in \{\mathrm{S}, \mathrm{I}, \mathrm{R}\}$ のいずれかであるとします (表 1)。状態 S の個体は、隣接するセルの一つでも状態 I の個体が居る場合に、確率 β で病気に罹り、状態 I になります。周囲の状態 I の個体数によって感染確率が変わらないことに注意します。また、状態 I の個体は、周囲のセルの状態にかかわらず、確率 γ で病気が治り、状態 R となります。状態 R となった個体は、免疫をもった状態ですから、状態は変化しません。個体の状態は同期的に更新するものとします。

システムのサイズを $n \times n$ とし、各時刻での状態 S、I、R の個体数を $S(t)$、$I(t)$、$R(t)$ とし、総個数 $N = S(t) + I(t) + R(t)$ は不変とします。時刻 $t = 0$ ででたらめに選んだセルに $I(0)$ 個の病気の個体と $S(0)$ 個の健康な個体をばらまき、シミュレーションを開始します。

シミュレーションの準備として、プロジェクト `Epidemic` を作成し、以下の URL から Java ソースファイルを取得し、プロジェクト `Epidemic` のソースファイルの下に置きます。また、ライブラリ `MyLib` も設定しましょう。

`http://http://aoba.cc.saga-u.ac.jp/lecture`
`/ModelingAndSimulation/javasrc/Epidemic/src.zip`

2　個体のクラス Individual

個体の状態を表すクラス Individual を定義します。個体の状態は、このクラスの中の列挙型として定義されています。列挙型とすることで、変数が他の値をとることを防ぐことができます。また、static public として定義することで、クラスのインスタンスを定義しなくても、クラスの外から状態の種類として使用することが可能となります。

```
//個体の状態
static public enum State {
    S, I, R;
}
private State state;//現在の状態
private State nextState;//次の状態
```

コンストラクタでは、最初の状態を与えます。

表 2: Individual クラスのコンストラクタ

コンストラクタと説明
Individual(State state) 初期の状態 state を与えて初期化する。

表 3: Individual クラスのメソッド

修飾子と型	メソッドと説明
State	getNextState(List<Individual> neighbours, double beta, double gamma, double r) 隣接個体のリスト neighbours、感染確率 beta、治癒確率 gamma、及び $[0, 1)$ の乱数 r を与えて、次の状態 nextState を求める。状態は更新しない。
State	getState() 現在の状態を取得する。
State	update() 次の状態 nextState を state に上書きすることで、状態更新を行う。

次の状態を求めるためには、周囲の個体の状態が必要です。周囲の個体のリスト neighbours を与えることで、次の状態を求めるメソッドが getNextState() です (表 3、Program 2.1)。このメソッドでは、内部で乱数を生成して用いるという方法ではなく、外から変数として乱数の値 r を与える形をとっています。こうすることで、このメソッドの応答を一意にすることができなす。非決定的な動作の部分は、このメソッドを呼ぶ際に与えられる乱数の値 r だけになります。

```
public State getNextState(List<Individual> neighbours,
        double beta, double gamma, double r) {
    nextState = state;
    switch (state) {//注目している個体の状態毎に判断
        case S://状態がSである場合
            boolean f = false;
            //周囲のセルに状態Iの個体が居るか
            for (Individual neighbou : neighbours) {
                f = f | (neighbou.state == State.I);
            }
            if (f && (r < beta)) {//病気に罹る


            }
            break;
        case I://状態がIである場合
            if (r < gamma) {//病気が治る


            }
            break;
        default:
            break;
    }
    return nextState;
}
```

Program 2.1: getNextState() メソッド

課題 1　Program 2.1 に示すメソッド `getNextState()` が、講義で説明した個体の状態変化の規則となるように、完成させなさい。

3　伝染病モデルのクラス EpidemicDynamics

クラス EpidemicDynamics は、CA 伝染病モデルを表すクラスです。

表 4: EpidemicDynamics クラスのコンストラクタ

コンストラクタと説明
EpidemicDynamics(int n, int nPop, int initI, double beta, double gamma) システムサイズ n (セル数は n×n)、個体数 nPop、初期の状態 I の個体数 initI、感染確率 beta、治癒確率 gamma を与えて初期化する。

3.1　初期配置の生成

```
/**
 * 0 から n*n-1 の整数をでたらめに並べ替えた数列を
 *
 * 入力配列として状態を初期化
 *
 * @param randomIntegers
 * @param initI 初期の状態 I の個体数
 * @param nPop 個体数の総数
 * @param nSite
 * @return
 */
static public Individual.State[] createRandomState(
        int randomIntegers[], int initI, int nPop, int nSite) {
    Individual.State states[] = new Individual.State[nSite];
    //初期の状態 I を initI 個配置
    for (int i = 0; i < initI; i++) {
        int k = randomIntegers[i];
        states[k] = Individual.State.I;
    }
    //初期の状態 S を nPop-initI 個配置


    return states;
}
```

Program 3.1: createRandomState() メソッド

コンストラクタを呼んだあと、メソッド initialize() を呼ぶことで、個体の初期配置を行います。その中で用いるランダムな初期配置を行うために、メソッド initialize() では、以下のような手順を行っています。まず、0 から $n^2 - 1$

表 5: EpidemicDynamics クラスのメソッド

修飾子と型	メソッドと説明
MacroState	countMacroState() 各状態の個体数の組 (MacroState クラス) として返す。
static Individual.State[]	createRandomState(int randomIntegers[], int initI, int nPop, int nSite) 0 から nSite-1 個の整数をでたらめに並べた配列 randomIntegers を与え、個体数 nPop、初期の状態 I の個体数 initI をでたらめに並べた状態の配列を返す。
double	getBeta() beta の値を返す。
double	getGamma() Gamma の値を返す。
int	getInitI() 初期の状態 I の個体数を返す。
int	getN() セル数を返す。
int	getnPop() 個体数を返す。
List<MacroState>	getSequence() 個体数の組 (MacroState クラス) の変化の列を返す。
Individual.State	getState(int x, int y) 指定した位置の固体の状態を返す。
void	initialize() 個体の状態を初期化する。
int	update() 状態を更新する。状態 I の個体数を返す。

の整数をでたらめに並べ替えた配列 randomIntegers を得ます。次に、メソッド setRandomState() で、メソッド createRandomState() を呼びます。この時、この配列 randomIntegers を使って、状態のランダムな列を生成し、セルに対応した状態を持つ個体を生成します。

課題 2　メソッド createRandomState() では、以下のようにランダムな状態の列を生成します。

1. 大きさ nSite(n^2) の状態の配列を作る。
2. 配列 randomIntegers の先頭から initI 個の値で指定される要素には、状態 I を置く。
3. 配列 randomIntegers のその後ろの nPop − initI 個の値、つまりインデクスが initI から nPop − 1 の値で指定される要素には、状態 S を置く。

Program 3.1 の、相当部分を記述し、完成させましょう。

3.2 状態更新

```
    public int update() {
        //各個体について、次の状態を計算
        for (int y = 0; y < n; y++) {
            for (int x = 0; x < n; x++) {
                int k = y * n + x;
                //隣接個体
                List<Individual> neighbours = getNeighbours(x, y);
                if (individuals[k] != null) {
                    individuals[k].getNextState(
                            neighbours, beta, gamma, Math.random());
                }
            }
        }
        //各個体の状態更新
        for (Individual ind : individuals) {
            if (ind != null) {
                ind.update();
            }
        }
        //状態 I と S の個体数を記録
        MacroState macroState = countMacroState();
        sequence.add(macroState);
        return macroState.I;
    }
```

Program 3.2: update() メソッド

系全体の状態更新を行うのが `update()` メソッドです (Program 3.2)。各位置 `(x,y)` のセルに対して、隣接するセルの個体一覧 `neighbours` を得て、個体の次の状態を `getNextState()` を用いて得ます。全てのセルの個体に対して、次の時刻の状態が得られたら、クラス `Individual` のメソッド `update()` で、状態を更新します。

4 単体テスト

ソフトウェア開発において、作成したプログラムの動作確認は欠かせません。その過程の一つに、単体テストがあります。単体テストは、部品の一つ一つの動作を確認する作業を指します。NetBeans では、JUnit を利用して、メソッドの動作を確認することができます。メソッド動作の確認では、実際のメソッドの戻り値と期待される値を比較することでテストを行います。

今回は、クラス Individual 中のメソッドの単体テストを実行しましょう。NetBeans の左側にあるプロジェクトウィンドウで、クラス Individual にマウスをあわせ、右ボタンを押します。そのメニュー中から「ツール」を選び、「テストの生成/更新」を選ぶと、新しいフォルダー test が作成され、テスト用クラスが生成されます。なお、このときに JUnit のバージョンを聞かれますから、4 を選んでください。IndividualTest クラスが生成されます。

デフォルトでは、IndividualTest クラスには、テスト実行の前処理と後処理を行う setUpClass()、tearDownClass()、setUp()、及び tearDown() と、各メソッドをテストするメソッドが生成されます。各メソッドのテストメソッドは、アノテーション@test で示されます。

今回は、getNextState() だけをテストしますから、以下の URL からダウンロードしたものと置き換えてください。

http://http://aoba.cc.saga-u.ac.jp/lecture
/ModelingAndSimulation/javasrc/Epidemic/test.zip

メソッド getNextState() をテストするのが testGetNextState1() ほか 4 つです。パラメタ β と γ を設定するとともに、周囲の個体の状態を List<Individual> neighbours で与えます。乱数を r で与えて、メソッド getNextState() を実行します。結果 result と、予想される結果 expResult を assertEquals() で比較しています。

課題 3　テストクラス IndividualTest に含まれる 4 つのテストの内容を記述しなさい。

課題 4　テストクラス IndividualTest を実行すると、テスト専用のウィンドウが開き、テストが実行され、テスト結果専用のウィンドウに出力されます。これらのテストは成功しましたか？失敗した場合には、getNextState() を訂正しましょう。

5　シミュレーション

クラス twoDCAGUI.EpidemicGUIMain を実行すると、指定したパラメタに対応したシミュレーションを実行します。各点は青が状態 S、赤が状態 I、水色が状態 R を表します。population を 0.7 とし

$$I_{\text{init}} = 0.1, \quad \beta = 0.1, \quad \gamma = 0.2$$

の時の動作を観測しましょう。「SAVE」ボタンを押すと、状態 I と状態 S の個体数の時間変化がファイル `out.txt` へ保存されます。gnuplot を使って、時間変化を見ましょう。

課題 5　population を 0.7 とし

$$I_{\text{init}} = 0.1, \quad \beta = 0.1, \quad \gamma = 0.1$$

及び

$$I_{\text{init}} = 0.1, \quad \beta = 0.1, \quad \gamma = 0.3$$

の時の動作を観測しましょう。状態 I の個体数の時間変化を、gnuplot を使って観察しましょう。

```
set terminal png enhanced 28
set xlabel "{/Italic t}"
set title "Epidemic Model"
set yrange [0:0.3]
set xrange [0:150]
set xtic 50
set output "epidemic-I.png"
set ytic 0.05
plot "Out.txt" using 1:2 with line lw 5 title "I"
```

Program 5.1: 状態 S と状態 I の個体数の時間変化を示す gnuplot スクリプトの例