10. セルオートマトン

2020/12/14

1 準備

講義で説明した、一次元セルオートマトンのシミュレーションの準備をします。NetBeans を開き、サンプルプログラムを取得します。また、必要に応じて、ライブラリとして、微分方程式の際に使用した MyLib を登録します。
`https://github.com/modeling-and-simulation-saga/CA`

2 クラス CA

2.1 状態更新規則

表 1　CA クラスのコンストラクタ

コンストラクタと説明
`CA(int n, int rule)` セルの総数 `n` とルール番号 `rule` を与えて初期化する。

ca パッケージ中のクラス CA は、一次元セルオートマトンを表すクラスです。周期境界条件、つまり系の両端がつながった鎖状のシステムとなるような境界条件を設定しています。各セルの状態を表す配列が、`cells[]` です。コンストラクタを表 1、メソッドを表 2 に示します。

時間発展は、3 個の連続するセルの状態に対して、中央のセルの次の時刻での状態を対応させることで決定します。3 個の隣接するセルの状態は 3 ビットで表現されるため、0 から 7 までの 8 通りです。3 ビットの入力に対して出力を表す配列が、`ruleMap[]` です。

課題 1　8 通りの入力に 0 または 1 を割り当てることでルールを定めます。つまり、ルールは 2^8 通り可能です。整数で表現されたルール番号 `rule` を二進数へ変換し、各ビットの値を `newRuleMap[]` という配列に割り当てるメソッドが `mkRuleMap()` です。メソッド `setRule()` では、この戻り値を `ruleMap` に代入します。

整数で与えられた `rule` の二進数表示の最下位ビットの値を `newRuleMap[0]`、次のビットの値を `newRuleMap[1]` と保存し、再上位ビットの値を `newRuleMap[7]` に保存します。`mkRuleMap()` 内に、この内容を記述しましょう。

課題 2　クラス CA を実行すると、rule-184 の場合について、`ruleMap[]` の内容を印刷します。実行して、正しい内容であることを確認しなさい。また、`main()` を変更し、rule-90 の場合について確認しなさい。

表 2　CA クラスのメソッド

修飾子と型	メソッドと説明
int[]	getCells() 現在のセルの状態を返す。
int	getN() セルの総数を返す。
int	getNumDifference() 状態更新によって値の変化したセルの数を返す。
int	getNumR() 初期化の際、値が 1 であったセルの数を返す。
int[]	getRuleMap() ruleMap を返す。
void	initialize() 確率 1/2 で、各セルの値を 1 とし、残りを 0 と初期化する。
void	initialize(double r) 確率 r で、各セルの値を 1 とし、残りを 0 と初期化する。
void	initializeSingle() 中央のセルの値を 1 とし、残りを 0 と初期化する。
static int[]	mkruleMap(int r) 与えられた整数の各ビット値を長さ 8 の配列に格納する。
static int	ruleMap2Int(int ruleMapDummy[]) ra[] を整数として返す。
static String	ruleMap2String(int ruleMapDummy[]) ra[] を文字列として返す。
void	setRule(int rule) ルール番号 rule を設定する。
String	state2String() セルの状態を文字列として返す。
int[]	update() 状態を更新し、最新のセルの状態を返す。

2.2　状態更新

次に、セルの状態更新を考えます。メソッド update() が状態更新のメソッドです。戻り値は、新しい状態を表します。セルの状態は配列 cells[] に保存されています。状態更新はすべてのセルに対して同時に行われるようにしなければなりません。もしも、セルの状態を端から更新すると、同時に更新した結果と異なってしまいます。そこで、次の時刻の状態を配列 cells[] に直接には書き込まず、ダミー配列 cellsDummy[] に一旦保存します。すべてのセルに対して計算が終わった後に、ダミー配列 cellsDummy[] から配列 cells[] に書き戻します。

各セルの次の状態を決めるには、その両隣のセルの状態を 3 ビットの整数に見立てて計算します。注目しているセルの番号を i とすると、$i \pm 1$ のセルの状態が必要です。システムは周期境界条件を有する、つまり輪のようにつながっていると考えていますから、セルの総数を n とすると、$i = 0$ の左隣は $n - 1$ 番のセル、$i = n - 1$ のセルの右隣は 0 番のセルとなります。そこで、セル i の左隣のセルの番号を l、右隣のセルの番号

を r とすると、次式で表すことができます。

$$l = (i - 1 + n) \bmod n \tag{2.1}$$
$$r = (i + 1) \bmod n \tag{2.2}$$

ここで mod n は n で除した余りを意味します。

課題 3　例えば $n = 100$ の場合、$i = 0$、$i = 50$ 及び $i = 99$ の両隣のセルが、式 (2.1) 及び (2.2) で求められることを確認しましょう。

課題 4　メソッド `update()` の部分を作成しましょう。

3　実行

配布したファイルに、パッケージの外に `CLIMain.java` というファイルがあります。これを実行すると、`output.txt` というテキストファイルの中に、シミュレーション結果を出力します。つまり、セルの状態が 1 の部分には*を、0 の部分には空白を出力します。VS Code などのテキストエディタを用いて、表示することができます。セルの総数が 100 となっていますから、100 文字が折り返されないように、エディタの設定を調整してください。

課題 5　配布ファイルでは、ルール番号は 90 になっています。シミュレーション結果を観察しなさい。また、ルール番号を 150 として観察しなさい。

4　単体テスト

動作が正しいことを確実に確認するための体系的な確認方法の一つが、単体テストです。単体テストは、関数などのプログラムの構成要素の動作を確認するテストです。Java では、JUnit というツールを使います。

今回はクラス `CA` の一部のメソッドの単体テストを実行しましょう。

1. 「プロジェクト」を表示し、クラス `CA` にマウスを合わせて、右ボタンを押します。
2. 現れたメニューから「ツール」→「テストの作成／更新」を選びます。
3. どの部分コードを生成するかを尋ねる画面が出ます (図 1)。ここでは、”Framework”を `JUnit4` にすることに注意してください。それ以外は、そのままにして「OK」を押します。
4. 「テストパッケージ」中にクラス `ca.CATest` が生成されます。いくつかのメソッドが自動で生成されます。

- `setUpClass()` メソッドは、このクラス内のテストの一番最初が実行される前に、一度だけ呼ばれます。
- `tearDownClass()` メソッドは、このクラス内のテストの一番最後が実行された後に、一度だけ呼ばれます。
- `setUp()` メソッドは、このクラス内の各テストが実行される前に呼ばれます。
- `tearDown()` メソッドは、このクラス内の各テストが実行された後に呼ばれます。

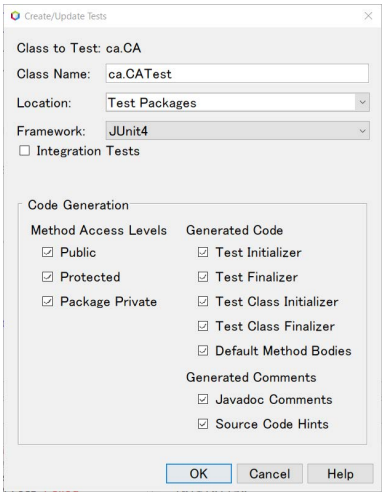

図 1　JUnit の設定画面。”Framework”を `JUnit4` にしていることに注意。

今回は、特に行うべきことがないので、上記の各メソッドはそのままにしておきます。この後ろに `test` で始まる、テストメソッドが続いています。

今回は `mkRuleMap` をテストする `testMkRuleMap()` だけを残して、他を削除します。また、このメソッド内の `fail()` メソッドも消してください。なお、この `fail()` メソッドは、テストを必ず失敗させるもので、テストメソッドを実装せずにおくことを防ぐ目的で置かれています。

`testMkRuleMap()` の中の `assertArrayEquals(expResult, result)` に注目します。このメソッドは `mkRuleMap()` によって得られた結果の配列 `result` と期待される結果の配列 `expResult` を比較するメソッドです。一致すればテスト結果を成功とし、一致していなければ失敗とします。

ソースコード 4.1　testMkRuleMap()

```
1    @Test
2    public void testMkRuleSet() {
3        System.out.println("mkRuleSet");
4        int rr = 90;
5        int[] expResult = {};//期待される値を記述
6        int[] result = CA.mkRuleMap(rr);
7        assertArrayEquals(expResult, result);
8    }
```

課題 6　rule-90 の場合をテストするように、結果として期待される配列 `expResult` を定義し、`testMkRuleMap()` 内を記述しなさい (ソースコード 4.1)。

課題 7　テストの実行には、プロジェクト画面でクラス `ca.CATest` を選び、マウス右ボタンで”Test File”を選びます。結果は、ソースファイルの下に”Test Result”という場所に出力します。正しく実行した場合には、緑色の結果が、失敗した場合は赤い結果が表示されます (図 2)。図 2) 中の緑のボタンをクリックすることで、繰り返しテストを実行することができます。

クラス `CA` には、ルールを表す配列を整数で返すメソッド `ruleMap2Int` があります。これを用いれば、整

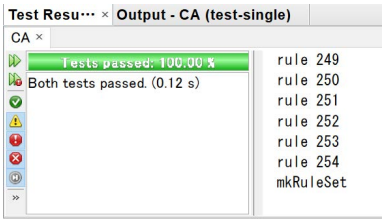

図 2　JUnit の設定画面

数で表されたルールを mkRuleMap を用いて配列へ一旦変換し、その配列を ruleMap2Int を用いて整数へ戻すことができます。もとの整数と ruleMap2Int の戻り値が等しいことを確かめることで、正しい動作であることをテストすることができます。

課題 8　整数で表されたルールが正しく変換されていることを確かめる、上述のテストに対応した新しいテストメソッド testRuleConv を作成しなさい (ソースコード 4.2)。ルールの整数は 1 から 254 まで変化させ、すべての場合について確かめなさい。なお、整数型の結果 result と期待する結果 expResult を比較するには、メソッド assertEquals(expResult,result) を用います。

ソースコード 4.2　testRuleConv()

```
1   @Test
2   public void testRuleConv(){
3       System.out.println("mkRuleset␣and␣ruleSet2Int");
4       for(int i=1;i<255;i++){//すべてのルールを確認
5           System.out.println("rule␣"+i);
6           int r[]=;//規則 i に対応した ruleMap
7           int j = ;//r[]を整数の規則番号へ戻したもの
8           assertEquals(i,j);
9       }
10  }
```

5　参考：processing を使ったシミュレーション実行

配布したサンプルプログラムのプロジェクトの直下に wolframCA というフォルダがあります。この下にある wolframCA.pde が processing のプログラムです。

必要な jar ファイル (CA.jar と MyLib.jar) を上記のフォルダ下の code に配置し、実行することで、時間発展の様子を観察することができます。前回同様、JDK-8 でビルドする必要があります。注意してください。また、wolframCA.pde 中のルール番号が記述されている場所を探し、ルール番号として 150 に変更してその動きを観察てみてください。