



|  |  |  |  |
| --- | --- | --- | --- |
|  | Cell 1 | Cell 3 | Cell 9 |
| Task 3 | 0.2553±0.44 | 0.0833±1.34 | 0±0 |
| Task 4 |  |  |  |

import java.util.Random;

import java.util.ArrayList;

import java.util.Arrays;

public class StochasticSystems {

private static int[][] s;

public static void main(String[] args) {

s = new int[][]{ // the states with the imaginary states, -1

{ -1, -1, -1, -1 },

{ -1, 1, 2, 3, -1 },

{ -1, 4, 5, 6, -1 },

{ -1, 7, 8, 9, -1 },

{ -1, -1, -1, -1 }

};

transitionProb();

steadyState();

}

/\*\*

\* probabilities based on task 2 or 1

\* @param x the x coord of a or b

\* @return the occupational probability of the given row

\*/

private static double occupationalProb(int x) {

switch(x) {

case 0 : // row 0 is a ghost state row

return 0;

case 1 :

return (1.0 / 6.0);

//return (1.0 / 9.0);

case 2 :

return (2.0 / 6.0);

//return (1.0 / 9.0);

case 3 :

return (3.0 / 6.0);

//return (1.0 / 9.0);

case 4 : // row 4 is a ghost state row

return 0;

default : // if somehow a number out of this range is passed, then we definitely

// don't want to go there

return 0;

}

}

/\*\*

\* calculates the acceptance probability

\* @param a pi(a)

\* @param b pi(b)

\* @return the probability of acceptance

\*/

private static double accProb(double a, double b) {

double p = 0; // p is our probability of acceptance

p = Math.min(1.0, (b/a));

return p;

}

/\*\*

\* calculate the acceptance probability for each pair

\* calculate the probability of choice, in this case the fixed value 1/4

\* multiply these two values

\* check if it adds to 1, if not, then add a self transition

\* For task one swap the occupationalProb to 1/9 in occupationalProb

\*/

private static void transitionProb() {

Random rng = new Random();

int[][] turtleACoords = new int[][] { { 1, 1 } };

int turtleA = 1; // state a

int[][] turtleBCoords = new int[][] { { 1, 1 } };

int turtleB = 1; // state b

for(int i = 0; i < s.length; i++) {

for(int j = 0; j < s[i].length; j++) {

turtleACoords[0][0] = i;

turtleACoords[0][1] = j;

if(s[i][j] != -1) { // ignore the ghost states

double[] accProb = new double[4];

double choiceProb = 1.0 / 4.0;

double[] transProb = new double[4]; // 1/4 \* accProb

// find the transition probabilities of NESW (0-3)

for(int k = 0; k < 4; k++) {

switch(k) {

case 0 : {

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

turtleBCoords[0][0] = x+1;

turtleBCoords[0][1] = y;

turtleB = s[turtleBCoords[0][0]][turtleBCoords[0][1]];

if(turtleB == -1) {

accProb[k] = accProb(occupationalProb(x), 0.0);

} else {

accProb[k] = accProb(occupationalProb(x), occupationalProb((x+1)));

}

transProb[k] = (accProb[k] \* choiceProb);

break;

}

case 1 : {

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

turtleBCoords[0][0] = x;

turtleBCoords[0][1] = y+1;

turtleB = s[turtleBCoords[0][0]][turtleBCoords[0][1]];

if(turtleB == -1) {

accProb[k] = accProb(occupationalProb(x), 0.0);

} else {

accProb[k] = accProb(occupationalProb(x), occupationalProb(x));

}

transProb[k] = (accProb[k] \* choiceProb);

break;

}

case 2 : {

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

turtleBCoords[0][0] = x-1;

turtleBCoords[0][1] = y;

turtleB = s[turtleBCoords[0][0]][turtleBCoords[0][1]];

if(turtleB == -1) {

accProb[k] = accProb(occupationalProb(x), 0.0);

} else {

accProb[k] = accProb(occupationalProb(x), occupationalProb((x-1)));

}

transProb[k] = (accProb[k] \* choiceProb);

break;

}

case 3 : {

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

turtleBCoords[0][0] = x;

turtleBCoords[0][1] = y-1;

turtleB = s[turtleBCoords[0][0]][turtleBCoords[0][1]];

if(turtleB == -1) {

accProb[k] = accProb(occupationalProb(x), 0.0);

} else {

accProb[k] = accProb(occupationalProb(x), occupationalProb(x));

}

transProb[k] = (accProb[k] \* choiceProb);

break;

}

}

}

double sum = 0;

for(int k = 0; k < transProb.length; k++) {

sum += transProb[k];

}

double selfTrans = (1 - sum);

if(selfTrans != 0) {

System.out.println("P(" + s[i][j] + "-> " + s[i][j] + ") = " + selfTrans);

}

for(int k = 0; k < transProb.length; k++) {

switch(k) {

case 0 :

if(transProb[k] == 0) {

} else {

System.out.println("P(" + s[i][j] + "-> " + s[i+1][j]

+ ") = " + transProb[k]);

}

break;

case 1 :

if(transProb[k] == 0) {

} else {

System.out.println("P(" + s[i][j] + "-> " + s[i][j+1]

+ ") = " + transProb[k]);

}

break;

case 2 :

if(transProb[k] == 0) {

} else {

System.out.println("P(" + s[i][j] + "-> " + s[i-1][j]

+ ") = " + transProb[k]);

}

break;

case 3 :

if(transProb[k] == 0) {

} else {

System.out.println("P(" + s[i][j] + "-> " + s[i][j-1]

+ ") = " + transProb[k]);

}

break;

}

}

}

}

}

}

private static void steadyState() {

// since the grid contains ghost states we need to go through and ignore them

// or we could just set it to 9, since we know the length..

int countLength = 0;

for(int i = 0; i < s.length; i++) {

for(int j = 0; j < s[i].length; j++) {

if(s[i][j] != -1) {

countLength++;

}

}

}

// initalise count[] as N+! and initalise all elements to 0

double[] count = new double[countLength+1];

for(int i = 0; i < count.length; i++) {

count[i] = 0;

}

Random rng = new Random();

int[][] turtleACoords = new int[][] { { 1, 1 } }; // x, y (i, j) coords

int turtleA = 1; // state a

int[][] turtleBCoords = new int[][] { { 1, 1 } };

int turtleB = 1; // state b

for(int n = 0; n < 10000; n++) { // number of repetitions

// reset a

turtleACoords[0][0] = 1;

turtleACoords[0][1] = 1;

turtleA = 1;

for(int m = 0; m < 3; m++) { // number of time steps, ignoring step 0 (since its just 1)

int k = 0; // number of transitions (including self)

double[] accProb = new double[4];

double choiceProb = 1.0 / 4.0;

double[] transProb = new double[4];

// find the transition probabilities of NESW (0-3)

for(int i = 0; i < 4; i++) {

switch(i) {

case 0 : {

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

turtleBCoords[0][0] = x+1;

turtleBCoords[0][1] = y;

turtleB = s[turtleBCoords[0][0]][turtleBCoords[0][1]];

if(turtleB == -1) {

accProb[i] = accProb(occupationalProb(x), 0.0);

} else {

accProb[i] = accProb(occupationalProb(x), occupationalProb((x+1)));

}

transProb[i] = (accProb[i] \* choiceProb);

break;

}

case 1 : {

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

turtleBCoords[0][0] = x;

turtleBCoords[0][1] = y+1;

turtleB = s[turtleBCoords[0][0]][turtleBCoords[0][1]];

if(turtleB == -1) {

accProb[i] = accProb(occupationalProb(x), 0.0);

} else {

accProb[i] = accProb(occupationalProb(x), occupationalProb(x));

}

transProb[i] = (accProb[i] \* choiceProb);

break;

}

case 2 : {

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

turtleBCoords[0][0] = x-1;

turtleBCoords[0][1] = y;

turtleB = s[turtleBCoords[0][0]][turtleBCoords[0][1]];

if(turtleB == -1) {

accProb[i] = accProb(occupationalProb(x), 0.0);

} else {

accProb[i] = accProb(occupationalProb(x), occupationalProb((x-1)));

}

transProb[i] = (accProb[i] \* choiceProb);

break;

}

case 3 : {

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

turtleBCoords[0][0] = x;

turtleBCoords[0][1] = y-1;

turtleB = s[turtleBCoords[0][0]][turtleBCoords[0][1]];

if(turtleB == -1) {

accProb[i] = accProb(occupationalProb(x), 0.0);

} else {

accProb[i] = accProb(occupationalProb(x), occupationalProb(x));

}

transProb[i] = (accProb[i] \* choiceProb);

break;

}

}

}

double sum = 0;

for(int i = 0; i < transProb.length; i++) {

sum += transProb[i];

}

double selfTrans = (1 - sum);

if(selfTrans != 0) { k++; }

for(int i = 0; i < transProb.length; i++) {

switch(i) {

case 0 :

if(transProb[i] != 0) { k++; }

break;

case 1 :

if(transProb[i] != 0) { k++; }

break;

case 2 :

if(transProb[i] != 0) { k++;}

break;

case 3 :

if(transProb[i] != 0) { k++; }

break;

}

}

// initialse t and put the transition value into it

double[] t = new double[k+1];

t[0] = 0;

for(int i = 1; i < t.length; i++) {

if(transProb.length >= i) {

if(transProb[i-1] != 0) {

t[i] = transProb[i-1];

for(int j = 1; j < t.length; j++) {

if(t[j] == 0) {

t[j] = transProb[i-1];

}

}

}

}

}

if(selfTrans != 0) { t[t.length-1] = selfTrans; }

Arrays.sort(t); // sort t in ascending order

for(int i=1; i < t.length/2; i++){ // flip the array into descending

double temp = t[i];

t[i] = t[t.length-i];

t[t.length-i] = temp;

}

double r = rng.nextDouble(); // r is between 0-1

double next = -1; // where we want to go next

double tSum = 0;

for(int i = 0; i < t.length; i++) {

if((tSum < r) && (r <= (tSum + t[i]))) {

next = t[i];

i = t.length; // end the loop after this iteration..

} else {

tSum += t[i];

}

}

int nextCounter = 0; // since the same values are ordered randomly, pick one randomly

ArrayList<Integer> nextTrans = new ArrayList<Integer>();

for(int i = 0; i < transProb.length; i++) {

if(next == transProb[i]) {

nextCounter++;

nextTrans.add(i);

}

}

int moveTo = -1;

if(nextCounter != 0) {

int random = rng.nextInt(nextCounter);

moveTo = nextTrans.get(random);

}

int x = turtleACoords[0][0];

int y = turtleACoords[0][1];

switch(moveTo) {

case 0 :

turtleACoords[0][0] = x+1;

turtleACoords[0][1] = y;

turtleA = s[turtleACoords[0][0]][turtleACoords[0][1]];

break;

case 1 :

turtleACoords[0][0] = x;

turtleACoords[0][1] = y+1;

turtleA = s[turtleACoords[0][0]][turtleACoords[0][1]];

break;

case 2 :

turtleACoords[0][0] = x-1;

turtleACoords[0][1] = y;

turtleA = s[turtleACoords[0][0]][turtleACoords[0][1]];

break;

case 3 :

turtleACoords[0][0] = x;

turtleACoords[0][1] = y-1;

turtleA = s[turtleACoords[0][0]][turtleACoords[0][1]];

break;

default :

// this is a self transition, do nothing..

break;

}

}

count[turtleA] += 1;

}

int n = 10000;

System.out.println();

System.out.println("Estimated prob of 1: " + (count[1]/n));

System.out.println("Estimated prob of 3: " + (count[3]/n));

System.out.println("Estimated prob of 9: " + (count[9]/n));

}

}