Experimenting With Two-Dimensional Turing Machines

A Turing Machine is, in essence, a finite state machine. Unlike the modern day computer, a Turing Machine has an infinite amount of memory, which is modelled by an infinitely long tape. A two-dimensional Turing Machine is a special case of a Turing Machine, where there are N reels of tape, all of which are infinitely long, that the Turing Machine can access.

Turing Machines follow a very simple routine: given some input, a simple operation is executed and then the active region of the tape is shifted, either left or right (or up and down in the case of the two-dimensional Turing Machine).

For the purposes of this assignment, a minor variation on the standard Turing Machine state transition guard is used. Typically, transitions are labelled as such:

\[
\text{Input/Output Tape Shift}
\]

Taking an example, a/b R, this says if an ‘a’ is read whilst in this state, then we shall replace that with a ‘b’ and shift the active region of the tape to the right. For this assignment, the only difference will be with the Tape Shift section of the transition guard. Rather than just a single left (L) or right (R) shift, up (U) and down (D) will also be specified. A typical transition will take the form of:

\[
\{ R\text{.G.B.} \}/\{ R\text{.G.B.} \} \{ L|R \}\{ U|D \}
\]

In plain English, the above rule states that for a given input colour, it will be replaced by an output colour, and the tape will be shifted either left or right and up or down.

A Java program was constructed to emulate a two-dimensional Turing Machine. This emulator is highly configurable, allowing the user to specify the size of the two dimensional array used to emulate the set of infinite tapes. Each tile in the grid is assigned a colour based on the pseudo-input that the user specifies. It is important to note that the user does not actually specify the input to the emulator; instead the user provides the emulator
with a pseudo-input which is interpreted and acted upon. The user can specify if the emulator will populate the grid with a random colour for every tile, or if the grid will be filled completely with black, white, red, green or blue. The emulator requires a 20 by 20 pixel square for each tile the user specifies.

The Turing Machine emulated in this program can be thought of as a two state machine with a large number of self-referential state transitions. Below is a short-hand notation of this Turing Machine’s diagrammatical representation. To draw in every possible transition would be near impossible.

The diagram above states that for a given colour \( \{r, g, b\} \) set, a function is applied to each colour component separately. These functions are labelled RED, GREEN and BLUE. Their Turing Machine representation is demonstrated in Appendix 3. The active region of the infinite tape set can be shifted in one of four combinations from the \( \{R, L, U, D\} \) set. This shift can be described via a Turing Machine as well, however a pseudo code representation is shown in Appendix 4. Further, for the Turing Machine to enter its final state, q1, it must find a tile where all the components of the colour are equal. Using the colour changing algorithm developed here in conjunction with the solid colour tile input, it should be noted that this Turing Machine will not halt. For the random input it is quite possible that it will halt, but it is not guaranteed.
The emulated Turing Machine begins at tile \( \{0,0\} \), or the first location of the first infinite tape reel. In strict Turing Machine terms, the colour of the tile is read in then three separate machines are applied to each component to determine what colour should be output. The shift algorithm is applied to the input to decide how to move the tapes. As modern day machines have physical memory constraints, if the tape reaches the end of physical memory, it is simply wrapped around to provide the illusion of an infinite tape.

\[\text{Figure 2: 10, 100, 1000 and 10000 transitions of the implemented Turing Machine on Black input}\]

The figure above shows four separate executions of the implemented Turing Machine on the black tile input with a 10 by 10 tile grid. Each quadrant shows a different number of transitions, we can see the current state (the red outlined tile), but we have no idea of the history. If one were to watch this Turing Machine for 100 transitions, one could discern the repetitive pattern that these input conditions produce. To show this pattern, a Turing Train was implemented. A Turing Train is an extension of the concept of the standard Turing Machine. It considers the current tile as the train engine and the previously visited \( n \) tiles as the carriages. Figure 3 below shows a 10 carriage Turing train on the same input with the same conditions. Due to a quirk in the Java paradigm, any given Swing component can only be added...
in one location. This is handy as it shows where the Turing Train has overlapped on its path (solid blue squares).

Figure 3: 100 transitions of the implemented Turing Train on the Black input

Figure 4 below shows a Turing Train on a random colour set. The interesting thing to note is the locality of reference the Train exhibits on a random set. The train is 100 carriages long, yet approximately half of the train is visible. This shows the Trains tendency to cluster about itself, rather than branch out.

Figure 4: 1000 transitions of a 100 carriage Turing Train on a random colour set

Program logic bugs aside, the most difficult part of this assignment was determining exactly what a Turing Machine is. It is fine to have a formal definition (like the one stated previously), but what exactly does it mean?
What does a Turing Machine look like? It is very easy to fall into a routine and deem everything that is presented as not ‘Turing Machineish’ enough. One must realise, however, that anything which can be computed on a modern day computer can also be executed on a Turing Machine in some way. Once this concept has been understood, it is a small step from coding up some algorithm (in this case a two-dimensional Turing Machine emulator) to defining the equivalent Turing Machine. It should be noted again that the equivalent Turing Machine is not the equivalent of the emulator itself, but of the Turing Machine which it is emulating.
import java.awt.*;
import javax.swing.*;
import java.util.*;

public class TuringMachine extends JFrame{
    private TuringMachineCell[][] grid;
    private JPanel selector;
    private int xsize;
    private int ysize;
    private int x;
    private int y;

    private TuringMachine(int generator, int xsize, int ysize) {
        Random r = new Random();
        this.grid = new TuringMachineCell[xsize][ysize];
        this.selector = new JPanel();
        this.x = 0;
        this.y = 0;
        this.xsize = xsize;
        this.ysize = ysize;

        switch(generator) {
            case 1:
                /** CASE 1: random colour generator **/
                // generate random colors for the grid
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(r.nextInt(256), r.nextInt(256), r.nextInt(256)));
                    }
                }
                break;
            case 2:
                /** CASE 2: black colour generator **/
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(0,0,0));
                    }
                }
                break;
            case 3:
                /** CASE 3: white colour generator **/
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(255,255,255));
                    }
                }
                break;
            case 4:
                /** CASE 4: red colour generator **/
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(255,0,0));
                    }
                }
                break;
            case 5:
                /** CASE 5: green colour generator **/
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(0,255,0));
                    }
                }
                break;
        }
    }
}
case 6:
    /** CASE 6: blue colour generator **/
    for(int i = 0; i < this.grid.length; i++) {
        for(int j = 0; j < this.grid[i].length; j++) {
            this.grid[i][j] = new TuringMachineCell(new Color(0,0,255));
        }
    }
    break;
}

// display options
this.setLayout(null);
this.setPreferredSize(new Dimension(xsize*20, ysize*20));

// place selector on the frame
this.selector.setLayout(null);
this.selector.setBounds(x, y, 30, 30);
this.selector.setBackground(Color.red);
this.getContentPane().add(this.selector);

// place the grid colors on the frame
for(int i = 0; i < this.grid.length; i++) {
    for(int j = 0; j < this.grid[i].length; j++) {
        this.grid[i][j].panel.setBounds(i*20+5, j*20+5, 20, 20);
        this.getContentPane().add(this.grid[i][j].panel);
    }
}

// remove current grid selection and add to selector
this.remove(this.grid[x][y].panel);
this.grid[x][y].panel.setBounds(5, 5, 20, 20);
this.selector.add(this.grid[x][y].panel);

// final display options
this.setSize(xsize*20 + 15, ysize*20 + 35);
this.setResizable(false);
this.setVisible(true);
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public boolean processCell() {
    Random rand = new Random();
    int color = this.grid[x][y].color.getRed();
    rand.setSeed(this.grid[x][y].color.getGreen());
    int horizontal = rand.nextInt(256);
    rand.setSeed(this.grid[x][y].color.getBlue());
    int vertical = rand.nextInt(256);

    // mod colour of current cell
    int r = (((color / 100) * 10) + color + 10) % 256;
    int g = (((color % 100) / 10) * 10) + horizontal + 10) % 256;
    int b = (((color % 100) % 10) * 10) + vertical + 10) % 256;
    this.grid[x][y].color = new Color(r, g, b);

    // mod x
    if(horizontal % 2 == 0) {
        this.x--;
        if(this.x < 0)
            this.x = xsize - 1;
    } else {
        this.x++;
        if(this.x > xsize - 1)
            this.x = 0;
    }
// mod y
if (vertical % 2 == 0) {
    this.y--;
    if (this.y < 0)
        this.y = ysize - 1;
} else {
    this.y++;
    if (this.y > ysize - 1)
        this.y = 0;
}

// remove all components
this.getContentPane().removeAll();

// reinsert components
this.selector.removeAll();
this.selector.setBounds(x*20, y*20, 30, 30);
this.getContentPane().add(this.selector);

// place the grid colors on the frame
for (int i = 0; i < this.grid.length; i++) {
    for (int j = 0; j < this.grid[i].length; j++) {
        this.grid[i][j].panel.setBounds(i*20+5, j*20+5, 20, 20);
        this.getContentPane().add(this.grid[i][j].panel);
    }
}

// place the grid colors on the frame
for (int i = 0; i < this.grid.length; i++) {
    for (int j = 0; j < this.grid[i].length; j++) {
        this.grid[i][j].panel.setBounds(i*20+5, j*20+5, 20, 20);
        this.getContentPane().add(this.grid[i][j].panel);
    }
}

// remove current grid selection and add to selector
this.remove(this.grid[x][y].panel);
this.grid[x][y].panel.setBounds(5, 5, 20, 20);
this.grid[x][y].refresh();
this.selector.add(this.grid[x][y].panel);

// refresh frame
this.getContentPane().validate();
this.getContentPane().repaint();

// check if colour is a shade of grey
if (r == g && r == b)
    return true;
return false;
}

private class TuringMachineCell {
    public JPanel panel;
    public Color color;

    public TuringMachineCell(Color color) {
        this.color = color;
        this.panel = new JPanel();
        this.panel.setBackground(this.color);
    }

    public void refresh() {
        this.panel.setBackground(this.color);
    }
}
public static void main(String[] args) throws Exception{
    if(args.length != 3) {
        System.out.println("Usage: java TuringMachine [generator] [x size] [y size]" );
        System.out.println(" [generator] : 1 = RANDOM");
        System.out.println(" [generator] : 2 = BLACK");
        System.out.println(" [generator] : 3 = WHITE");
        System.out.println(" [generator] : 4 = RED");
        System.out.println(" [generator] : 5 = GREEN");
        System.out.println(" [generator] : 6 = BLUE");
        System.exit(1);
    }
    TuringMachine tm = new TuringMachine(
        Integer.parseInt(args[0]),
        Integer.parseInt(args[1]),
        Integer.parseInt(args[2]));

    while(!tm.processCell()) {
        // for human eyes
        Thread.sleep(10);
    }

    while(true) {
        //do nothing
    }
}
import java.awt.*;
import javax.swing.*;
import java.util.*;

public class TuringTrain extends JFrame{
    private TuringMachineCell[][] grid;
    private JPanel engine;
    private TuringMachineCarriage[] carriage;
    private int xsize;
    private int ysize;
    private int x;
    private int y;

    private TuringTrain(int generator, int xsize, int ysize, int carriage) {
        Random r = new Random();
        this.grid = new TuringMachineCell[xsize][ysize];
        this.engine = new JPanel();
        this.carriage = new TuringMachineCarriage[carriage];
        this.x = 0;
        this.y = 0;

        this.xsize = xsize;
        this.ysize = ysize;

        // initialise carriages
        for(int i=0; i < carriage; i++)
            this.carriage[i] = new TuringMachineCarriage();

        switch(generator) {
            case 1:
                /** CASE 1: random colour generator **/
                // generate random colors for the grid
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(
                            r.nextInt(256), r.nextInt(256), r.nextInt(256)));
                    }
                }
                break;
            case 2:
                /** CASE 2: black colour generator **/
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(0,0,0));
                    }
                }
                break;
            case 3:
                /** CASE 3: white colour generator **/
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(255,255,255));
                    }
                }
                break;
            case 4:
                /** CASE 4: red colour generator **/
                for(int i = 0; i < this.grid.length; i++) {
                    for(int j = 0; j < this.grid[i].length; j++) {
                        this.grid[i][j] = new TuringMachineCell(new Color(255,0,0));
                    }
                }
                break;
            case 5:
        }
    }
}
/** CASE 5: green colour generator **/
for(int i = 0; i < this.grid.length; i++) {
    for(int j = 0; j < this.grid[i].length; j++) {
        this.grid[i][j] = new TuringMachineCell(new Color(0,255,0));
    }
}
break;
case 6:
/** CASE 6: blue colour generator **/
for(int i = 0; i < this.grid.length; i++) {
    for(int j = 0; j < this.grid[i].length; j++) {
        this.grid[i][j] = new TuringMachineCell(new Color(0,0,255));
    }
}
break;
}
// display options
this.setLayout(null);
this.setPreferredSize(new Dimension(xsize*20, ysize*20));

// place engine on the frame
this.engine.setLayout(null);
this.engine.setBounds(x, y, 30, 30);
this.engine.setBackground(Color.red);
this.getContentPane().add(this.engine);

// place the grid colors on the frame
for(int i = 0; i < this.grid.length; i++) {
    for(int j = 0; j < this.grid[i].length; j++) {
        this.grid[i][j].panel.setBounds(i*20+5, j*20+5, 20, 20);
        this.getContentPane().add(this.grid[i][j].panel);
    }
}

// remove current grid selection and add to engine
this.remove(this.grid[x][y].panel);
this.grid[x][y].panel.setBounds(5, 5, 20, 20);
this.engine.add(this.grid[x][y].panel);

// final display options
this.setSize(xsize*20 + 15, ysize*20 + 35);
this.setResizable(false);
this.setVisible(true);
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public boolean processCell() {
    Random rand = new Random();
    int color = this.grid[x][y].color.getRed();
    rand.setSeed(this.grid[x][y].color.getGreen());
    int horizontal = rand.nextInt(256);
    rand.setSeed(this.grid[x][y].color.getBlue());
    int vertical = rand.nextInt(256);

    // mod colour of current cell
    int r = (((color / 100) * 10) + color + 10) % 256;
    int g = (((color % 100) / 10) * 10) + horizontal + 10) % 256;
    int b = (((color % 100) % 10) * 10) + vertical + 10) % 256;
    this.grid[x][y].color = new Color(r, g, b);

    // save x and y
    int ecs = this.x;
    int why = this.y;

    // mod x
    if(horizontal % 2 == 0) {
        this.x--;
    }
if(this.x < 0)
  this.x = xsize - 1;
} else {
  this.x++;
  if(this.x > xsize - 1)
    this.x = 0;
}

// mod y
if(vertical % 2 == 0) {
  this.y--;
  if(this.y < 0)
    this.y = ysize - 1;
} else {
  this.y++;
  if(this.y > ysize - 1)
    this.y = 0;
}

// remove all components
this.getContentPane().removeAll();

// reinsert components
this.engine.removeAll();
this.engine.setBounds(x*20, y*20, 30, 30);
this.getContentPane().add(this.engine);

// update carriages
for(int i = 0; i < carriage.length - 1; i++) {
  this.carriage[i].panel.removeAll();
  this.carriage[i].x = this.carriage[i+1].x;
  this.carriage[i].y = this.carriage[i+1].y;
  this.carriage[i].panel.setBounds(this.carriage[i].x*20,
       this.carriage[i].y*20, 30, 30);
  this.getContentPane().add(this.carriage[i].panel);
}

this.carriage[carriage.length - 1].x = ecs;
this.carriage[carriage.length - 1].y = why;
this.getContentPane().add(this.carriage[carriage.length - 1].panel);

// place the grid colors on the frame
for(int i = 0; i < this.grid.length; i++) {
  for(int j = 0; j < this.grid[i].length; j++) {
    this.grid[i][j].panel.setBounds(i*20+5, j*20+5, 20, 20);
    this.getContentPane().add(this.grid[i][j].panel);
  }
}

// remove train selection and add to carriage
for(int i = 0; i < carriage.length; i++) {
  this.remove(this.grid[carriage[i].x][carriage[i].y].panel);
  this.grid[carriage[i].x][carriage[i].y].panel.setBounds(5, 5, 20, 20);
  this.grid[carriage[i].x][carriage[i].y].refresh();
  this.carriage[i].panel.add(this.grid[carriage[i].x][carriage[i].y].panel);
}

// remove current grid selection and add to engine
this.remove(this.grid[x][y].panel);
this.grid[x][y].panel.setBounds(5, 5, 20, 20);
this.grid[x][y].refresh();
this.engine.add(this.grid[x][y].panel);
// refresh frame
this.getContentPane().validate();
this.getContentPane().repaint();

// check if colour is a shade of grey
if(r == g && r == b)
    return true;
return false;
}

private class TuringMachineCell {
    public JPanel panel;
    public Color color;

    public TuringMachineCell(Color color) {
        this.color = color;
        this.panel = new JPanel();
        this.panel.setBackground(this.color);
    }

    public void refresh() {
        this.panel.setBackground(this.color);
    }
}

private class TuringMachineCarriage {
    public JPanel panel;
    public int x;
    public int y;

    public TuringMachineCarriage() {
        this.panel = new JPanel();
        this.panel.setLayout(null);
        this.panel.setBackground(Color.blue);
        this.x = 0;
        this.y = 0;
    }
}

public static void main(String[] args) throws Exception{
    if(args.length != 4) {
        System.out.println("Usage: java TuringMachine [generator] [x size] [y size] " + "[train length]");
        System.out.println("[generator] : 1 = RANDOM");
        System.out.println("[generator] : 2 = BLACK");
        System.out.println("[generator] : 3 = WHITE");
        System.out.println("[generator] : 4 = RED");
        System.out.println("[generator] : 5 = GREEN");
        System.out.println("[generator] : 6 = BLUE");
        System.exit(1);
    }

    TuringTrain tm = new TuringTrain(
        Integer.parseInt(args[0]),
        Integer.parseInt(args[1]),
        Integer.parseInt(args[2]),
        Integer.parseInt(args[3])
    );

    while(!tm.processCell()) {
        // for human eyes
        Thread.sleep(10);
    }

    while(true) {
        // do nothing
    }
}
Appendix 3: RED GREEN And BLUE Turing Machine Representations

**cd RED**

Takes an input string of the following form:
B r B r

**cd GREEN**

Takes an input string of the following form:
B r B RAND(g)

**cd BLUE**

Takes an input string of the following form:
B r B RAND(b)
Appendix 4: Active Region Tape Shift Algorithm

/* obtain a pseudo-random number, using the GREEN component as the
   seed with an upper bound of 256 */
horizontal = RANDOM(green, 256);

/* obtain a pseudo-random number, using the BLUE component as the
   seed with an upper bound of 256 */
vertical = RANDOM(blue, 256);

IF(EVEN(horizontal) AND EVEN(vertical))
   SHIFT(right, down);

IF(EVEN(horizontal) AND ODD(vertical))
   SHIFT(right, up);

IF(ODD(horizontal) AND EVEN(vertical))
   SHIFT(left, down);

IF(ODD(horizontal) AND ODD(vertical))
   SHIFT(left, up);