Programming Systems and Environments – Lab 6

Jakub Grzana, 241530

Task 1 – Customers and Clerks

First I've implemented two classes to deal with multithreading: MyThread and MyThreadAnchor. Anchor is abstract class with Run() function, so it's the same as Runnable (after consideration, I should've use this template instead…) but it also allows underlying code to throw exceptions, so it's a bit more convenient for me. MyThread allows to create new thread and run referred class within it. So after implementing this, I didn't have to worry about multithreading code anymore.

Then I've implemented Customer class, that has built-in random number generator (normal distribution) and that stores time this customer will require to be serviced. Default constructor of this class generates this time.

Next stop, Queue. It builds upon ConcurrentLinkedQueue, Aside from implementing push() and pop() functions (to add/remove from queue) it keeps track of accepted, serviced or rejected customers. Queue has arbitrary limit of customers.

Next class, CustomerGenerator. It extends MyThreadAnchor and has built-in RNG realizing exponential distribution, and reference to Queue. It generates random number of seconds and sleeps for this duration, then adds new Customer to Queue.

And finally, Clerk, which also have reference to Queue and extends MyThreadAnchor. In endless loop, it listens for new customers in queue and if it receives customer, then it waits for given number of seconds.

Task 2 – Hilbert Maze and A* algorithm

There might be easier solution to this problem, but it was actually interesting for me, so I've decided to implement Hilbert Maze class and A* algorithm for pathfinding in this maze. My solution isn't clean one, but it works.

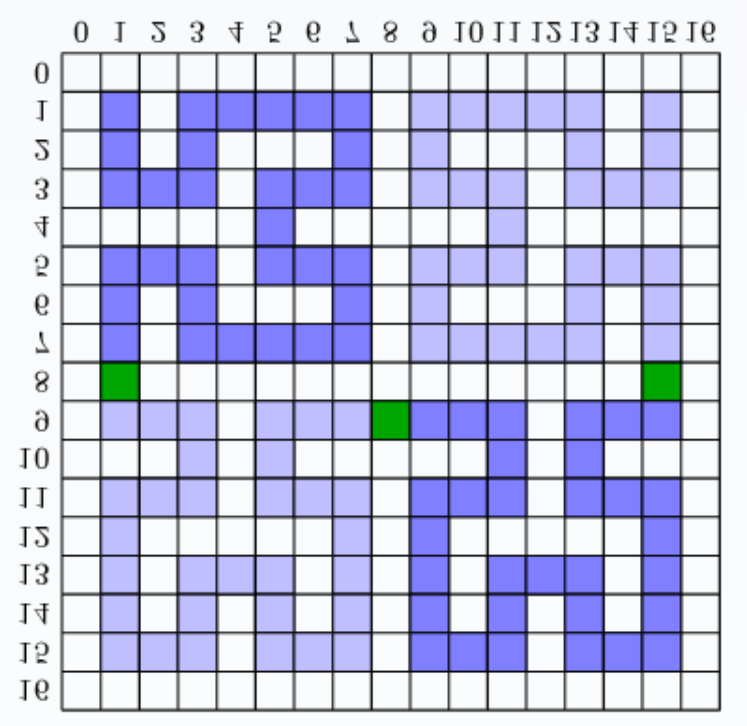
I've implemented Node class that represents single cell within maze. It has weight and visualization attribute. Visualization is character that is used when printing maze, weight can store any number, it is later used for pathfinding. Maximal value of weight is treated as "impassable".

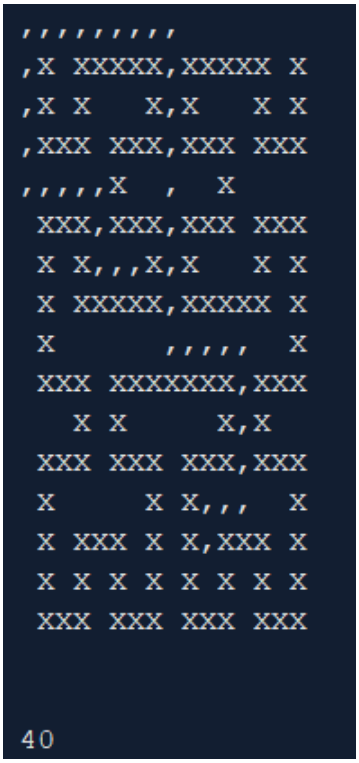
Another class, Point, is used for A* algorithm cause it requires more information about position than Node provides. It stores G, H and F of given point in a maze.

Finally, HilbertMaze class. This one is giant and it stores the maze itself, as well as providing tools for creation of Hilbert Maze. It can create maze for any positive number of N, it does so by keeping maze for N=1 in memory, and generating mazes for higher N recursively with rotations and pasting. HilbertMaze allows also to mark specific nodes

with custom symbols, this can be used to mark a path found by A* algorithm (that is also implemented within Hilbert Maze class)

Quick note about coordinates: maze is stored within ArrayList<ArrayList<Node>> with notation that in C could be written as: maze[row][column]. It was easier for me to create mazes with following structure:





ManhattanDistance was used as heuristic function.

Conclusions

For better code separation, I've implemented Task 1 and 2 in separate Java packages.

I think my code for Task 1 is good multithreading Java code, and pretty well done in OOP paradigm. Task itself was kinda dull, but it served its purpose.

Implementation of Hilbert Maze and A* algorithm in Java was hard but rewarding task. My biggest mistake in coding was probably not creating setter/getter function for nodes within that maze, which makes code more hard to read than it's necessary.

The code works. I'm attaching it in zipfile, cause I believe this time it will be way easier to read for you in IDE instead of paper. Those tasks were pretty complicated and there're many files.

Bibliography:

Pretty much none, I've used only things from previous lab (to implement MyThread), documents provided by teacher (to implement Hilbert Maze) and my own knowledge (to implement A* algorithm, I did it in the past)

Normal distribution in java [access 17.05.2022]
https://stackoverflow.com/questions/29389412/generating-numbers-which-follow-normal-distribution-in-java

Exponential distribution in java [access 17.05.2022]
https://stackoverflow.com/questions/29020652/java-exponential-distribution

CODE: Multithreading.java

```
package com.mycompany.multithreading;
import com.mycompany.multithreading.HilbertMaze.*;
import java.util.ArrayList;
import com.mycompany.multithreading.customerclerks.*;
import java.util.concurrent.TimeUnit;

public class Multithreading {

   // calculate weight (length) of path in Hilbert Maze
   public static int Accum(ArrayList<Point> path)
   {
      int output = 0;
      for(Point p : path)
      {
         output = output + p.Node().Weight();
      }
      return output;
   }

   public static void HilbertMazeExample()
   {
      HilbertMaze maze = HilbertMaze.Generate(3);
      ArrayList<Point> path = maze.Astar(6, 6, 13, 10);
      maze.Mark(path, ',');
      System.out.println(maze);
      System.out.println(Accum(path));
   }

   public static void CustomersClerksExample()throws Exception
   {
      final int clerk_num = 3;
      Queue queue = new Queue(10);
      ArrayList<MyThread> threads = new ArrayList<>();
      threads.add(new MyThread(new CustomerGenerator(queue)));
      for(int i = 0; i < clerk_num; ++i) { threads.add(new MyThread(new Clerk(queue))); }
      while(true)
      {
         TimeUnit.SECONDS.sleep(5);
```

```
            System.out.println(queue);
        }
    }

    public static void main(String[] args) throws Exception
    {
        HilbertMazeExample();
        CustomersClerksExample();
    }
}
```

CODE: Node.java

```
package com.mycompany.multithreading.HilbertMaze;
public class Node {
    private int weight = 0;
    private char visualisation = ' ';

    public final  boolean isPassable()
    {
        return this.weight != Integer.MAX_VALUE;
    }

    public int Weight()
    {
        return this.weight;
    }

    public Node(boolean passable)
    {
        this.weight = passable ? 1 : Integer.MAX_VALUE;
        this.visualisation = this.isPassable() ? ' ' : 'X';
    }


    public Node(int weight)
    {
        this.weight = weight;
        this.visualisation = this.isPassable() ? ' ' : 'X';
    }

    public void Mark(char visual)
    {
        this.visualisation = visual;
    }

    @Override public String toString()
    {
        return Character.toString(this.visualisation);
    }
}
```

CODE: Point.java

```
package com.mycompany.multithreading.HilbertMaze;
import java.lang.Math;
import java.util.ArrayList;
import java.util.*;

public class Point {
   final private int row;
   final private int column;
   final private Node node;

   private int g = 0;
   private int h = 0;
   private int f = 0;
   private Point previous = null;

   public static int ManhattanDistance(Point pos, int target_row, int target_col)
   {
      return Math.abs(pos.Row() - target_row) + Math.abs(pos.Column() - target_col);
   }

   public Point(int r, int c, Node n, Point prev, int target_row, int target_col)
   {
      this.row = r;
      this.column = c;
      this.node = n;
      this.previous = prev;

      this.g = prev != null ? prev.g + prev.node.Weight() : 0;
      this.h = ManhattanDistance(this, target_row, target_col);
      this.f = this.g + this.h;
   }

   public int Row() { return this.row; }
   public int Column() { return this.column; }
   public Node Node() { return this.node; }

   public void Mark(char visual)
   {
      this.node.Mark(visual);
   }

   public void Update(Point prev, int target_row, int target_col)
   {
      if(this.previous.g > prev.g)
      {
         this.previous = prev;
         this.g = prev.g + this.node.Weight();
         this.h = ManhattanDistance(this, target_row, target_col);
         this.f = this.g + this.h;
      }
   }
```

```
    public ArrayList<Point> GetPathTo()
    {
        ArrayList<Point> output = new ArrayList<>();
        for(Point iter = this; iter != null; iter = iter.previous)
        {
            output.add(iter);
        }
        Collections.reverse(output);
        return output;
    }

    public int GetF() { return this.f; }
    public int GetH() { return this.h; }

    @Override public boolean equals(Object v) {
        if(v instanceof Point)
        {
            Point ptr = (Point) v;
            if(this.row == ptr.row && this.column == ptr.column) return true;
        }
        return false;
    }

    @Override public String toString()
    {
        return String.format("(%d,%d)", this.row, this.column);
    }
}
```

CODE: HilbertMaze.java (I'm sorry sir, and goodluck)

```
package com.mycompany.multithreading.HilbertMaze;

import java.util.ArrayList;
import java.lang.Math;
import java.util.Collections;
import java.util.Comparator;

public class HilbertMaze {
    private ArrayList<ArrayList<Node>> maze = new ArrayList<>(); // maze[row][column]

    public int Size()
    {
        return this.maze.size();
    }

    static private int _SizeForN(int n)
    {
        return (int)Math.pow(2,(n+1)) + 1;
    }

    private void _Truncate(int n)
    {
        this.maze.clear();
        for(int r = 0; r < _SizeForN(n); ++r)
```

```
    {
        this.maze.add(new ArrayList<>());
        for(int c = 0; c < _SizeForN(n); ++c)
        {
            this.maze.get(r).add(new Node(true));
        }
    }
}

private void _Paste(int row_offset, int col_offset, HilbertMaze submaze)
{
    for(int sub_row = 0; sub_row < submaze.Size(); ++sub_row)
    {
        for(int sub_col = 0; sub_col < submaze.Size(); ++sub_col)
        {
            int row = sub_row + row_offset;
            int col = sub_col + col_offset;
            Node obj = submaze.maze.get(sub_row).get(sub_col);
            this.maze.get(row).set(col, obj);
        }
    }
}

static private HilbertMaze _Initial() // działa
{
    ArrayList<ArrayList<Node>> maze = new ArrayList<>();
    for(int r = 0; r < 5; ++r)
    {
        maze.add(new ArrayList<>());
        for(int c = 0; c < 5; ++c)
        {
            if(r>=1 && r <= 3 && c>=1 && c<=3)
            {
                Node obj = (c == 2 && r != 3) ? new Node(true) : new Node(false);
                maze.get(r).add(obj);
            }
            else
            {
                maze.get(r).add(new Node(true));
            }
        }
    }
    HilbertMaze output = new HilbertMaze();
    output.maze = maze;
    return output;
}

static public HilbertMaze Generate(int n)
{
    if(n == 1)
    {
        return _Initial();
    }

    HilbertMaze output = new HilbertMaze();
```

```
        output._Truncate(n);

        HilbertMaze topleft = new HilbertMaze(Generate(n-1));
        HilbertMaze topright = new HilbertMaze(topleft);
        HilbertMaze bottomleft = new HilbertMaze(topleft.Rotate(true));
        HilbertMaze bottomright = new HilbertMaze(topleft.Rotate(false));

        output._Paste(0, 0, bottomleft);
        output._Paste(0, _SizeForN(n)/2, bottomright);
        output._Paste(_SizeForN(n)/2, _SizeForN(n)/2, topright);
        output._Paste(_SizeForN(n)/2, 0, topleft);

        output.maze.get(_SizeForN(n)/2 + 1).set(_SizeForN(n)/2, new Node(false));
        output.maze.get(_SizeForN(n)/2).set(1, new Node(false));
        output.maze.get(_SizeForN(n)/2).set(_SizeForN(n) - 2, new Node(false));

        return output;
    }

    public HilbertMaze Rotate(boolean clockwise) // działa
    {
        ArrayList<ArrayList<Node>> output_maze = new ArrayList<>();
        for(int row = 0; row < this.maze.size(); ++row)
        {
            output_maze.add(new ArrayList<>());
            for(int column = 0; column < this.maze.get(row).size(); ++column)
            {
                Node obj = clockwise ? this.maze.get(column).get(row) : this.maze.get(this.maze.size() - 1 -
column).get(row);
                output_maze.get(row).add(obj);
            }
        }
        HilbertMaze output = new HilbertMaze();
        output.maze = output_maze;
        return output;
    }

    private HilbertMaze() {} // hiding constructor

    @Override public String toString() // działa
    {
        String output = "";
        for(ArrayList<Node> row : this.maze)
        {
            for(Node n : row)
            {
                output = output + n;
            }
            output = output + "\n";
        }
        return output;
    }
```

```
public void Mark(ArrayList<Point> path, char visual)
{
   for(Point p : path)
   {
      int row = p.Row();
      int col = p.Column();
      this.maze.get(row).get(col).Mark(visual);
   }
}

public HilbertMaze(HilbertMaze tocopy)
{
   for(int r = 0; r < tocopy.Size(); ++r)
   {
      this.maze.add(new ArrayList<>());
      for(int c = 0; c < tocopy.maze.get(r).size(); ++c)
      {
         Node obj = tocopy.maze.get(r).get(c);
         Node n = new Node(obj.Weight());
         this.maze.get(r).add(n);
      }
   }
}

public ArrayList<Point> Astar(int start_row, int start_col, int target_row, int target_col)
{
   if(!this.maze.get(target_row).get(target_col).isPassable()) return null;
   ArrayList<Point> open_nodes = new ArrayList<>();
   open_nodes.add(new Point(start_row,
         start_col,
         this.maze.get(start_row).get(start_col),
         null,
         target_row,
         target_col
   ));
   ArrayList<Point> closed_nodes = new ArrayList<>();

   while(true)
   {
      if(open_nodes.isEmpty()) break;
      Collections.sort(open_nodes, new Comparator<Point>(){
         @Override public int compare(Point a, Point b){
            return a.GetF() - b.GetF();
         }
     });

      ArrayList<Point> lowest_f = new ArrayList<>();
      for(Point p : open_nodes)
      {
         if(p.GetF() != open_nodes.get(0).GetF()) break;
         lowest_f.add(p);
      }
      Collections.sort(lowest_f, new Comparator<Point>(){
         @Override public int compare(Point a, Point b){
            return a.GetH() - b.GetH();
```

```
            }
        });

        Point current_node = lowest_f.get(0);

        if((current_node.Row() == target_row) && (current_node.Column() == target_col))
            return current_node.GetPathTo();

        open_nodes.remove(current_node);
        closed_nodes.add(current_node);

        for(int offset_row = -1; offset_row <= 1; ++offset_row)
        {
            for(int offset_col = -1; offset_col <= 1; ++offset_col)
            {
                if(Math.abs(offset_row) == Math.abs(offset_col)) continue;
                int pos_row = current_node.Row() + offset_row;
                int pos_col = current_node.Column() + offset_col;
                if(pos_row < 0 || pos_row >= this.Size()) continue;
                if(pos_col < 0 || pos_col >= this.Size()) continue;
                Node n = this.maze.get(pos_row).get(pos_col);
                Point neighbour = new Point(pos_row, pos_col, n, current_node, target_row, target_col);
                if(closed_nodes.contains(neighbour) || !neighbour.Node().isPassable()) continue;

                if(open_nodes.contains(neighbour))
                {
                    Point p = open_nodes.get(open_nodes.indexOf(neighbour));
                    p.Update(current_node, target_row, target_col);
                }
                else
                {
                    open_nodes.add(neighbour);
                }
            }
        }
    }
    return null;
  }
}
```

CODE: MyThreadAnchor.java

```
package com.mycompany.multithreading.customerclerks;

abstract public class MyThreadAnchor {
   abstract public void Run() throws Exception;
}
```

CODE: MyThread.java

```
package com.mycompany.multithreading.customerclerks;

public class MyThread implements Runnable {
   final private Thread t;
   final MyThreadAnchor obj;

   public MyThread(MyThreadAnchor obj)
   {
      this.obj = obj;
      this.t = new Thread(this);
      this.t.start();
   }

   @Override public void run()
   {
      try { this.obj.Run(); }
      catch (Exception e) {
       System.out.println("Thread interrupted: " + e);
    }
   }

   public void join() throws Exception
   {
      this.t.join();
   }
}
```

CODE: Customer.java

```
package com.mycompany.multithreading.customerclerks;
import java.util.Random;

class RandomGaussian
{
   final private double mean;
   final private double deviation;
   final private Random generator = new Random();
   public RandomGaussian(double mean, double deviation)
   {
      this.mean = mean;
      this.deviation = deviation;
   }
   public double next()
   { return this.mean + this.generator.nextGaussian()*this.deviation; }
}

public class Customer {
   private double time = 0; // given in seconds
   final private RandomGaussian generator = new RandomGaussian(0.15, 0.05); // mean, deviation given in
seconds
   Customer() { this.time = this.generator.next(); }
   double GetTime() { return this.time; }
}
```

CODE: Queue.java

```
package com.mycompany.multithreading.customerclerks;
import java.util.concurrent.*;

public class Queue {
   final private ConcurrentLinkedQueue<Customer> queue = new ConcurrentLinkedQueue<>();
   final int limit;

   int serviced = 0;
   int rejected = 0;
   int received = 0;

   public Queue(int limit) { this.limit = limit; }

   public boolean push(Customer ob) {
      if( this.queue.size() < this.limit)
      {
         this.received = this.received + 1;
         this.queue.add(ob);
         return true;
      }
      else
      {
         this.rejected = this.rejected + 1;
         return false;
      }
   }

   public Customer pop()
   {
      Customer out = this.queue.poll();
      if(out != null)
      { this.serviced = this.serviced + 1; }
      return out;
   }

   @Override public String toString()
   {
      return String.format("Serviced %d Waiting %d Rejected %d", this.serviced, this.received -
this.serviced, this.rejected);
   }
}
```

CODE: CustomerGenerator.java

```
package com.mycompany.multithreading.customerclerks;
import java.lang.Math;
import java.util.Random;
import java.util.concurrent.TimeUnit;

class RandomExp
{
   final private double lambda;
   final private Random generator = new Random();
   RandomExp(double lambda) { this.lambda = lambda; }
   public double get()
   {
      double u = this.generator.nextDouble();
      return Math.log(1-u) / (-this.lambda);
   }
}

public class CustomerGenerator extends MyThreadAnchor {
   final private RandomExp generator = new RandomExp(1.5); // lambda - given in seconds
   final private Queue ref_to_queue;
   public CustomerGenerator(Queue ref_to_queue) { this.ref_to_queue = ref_to_queue; }
   @Override public void Run() throws Exception
   {
      while(true)
      {
         long time = (long)this.generator.get() * 1000;
         TimeUnit.MILLISECONDS.sleep(time);
         this.ref_to_queue.push(new Customer());
      }
   }
}
```

CODE: Clerk.java

```
package com.mycompany.multithreading.customerclerks;
import java.util.concurrent.TimeUnit;

public class Clerk extends MyThreadAnchor {
   final private Queue ref_to_queue;
   public Clerk( Queue ref_to_queue) { this.ref_to_queue = ref_to_queue; }

   @Override public void Run() throws Exception
   {
      while(true)
      {
         Customer obj = this.ref_to_queue.pop();
         if(obj != null)
         {
            TimeUnit.MILLISECONDS.sleep((long)obj.GetTime()*1000);
         }
         TimeUnit.MILLISECONDS.sleep(100);
      }
   }
}
```