ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 458E
Functional Programming

Homework 2

Ali Emre Kaya
150210097
kayaemr21@itu.edu.tr

SPRING 2025

Contents

1 Schelling's Model of Segregation

Schelling's model of segregation is an agent-based model developed by economist Thomas Schelling [1]. In the Model, two different classes are separated from each other according to certain tolerance values and try to be happy among themselves, those who are not happy continue to take steps randomly until they find a place where they can be happier.

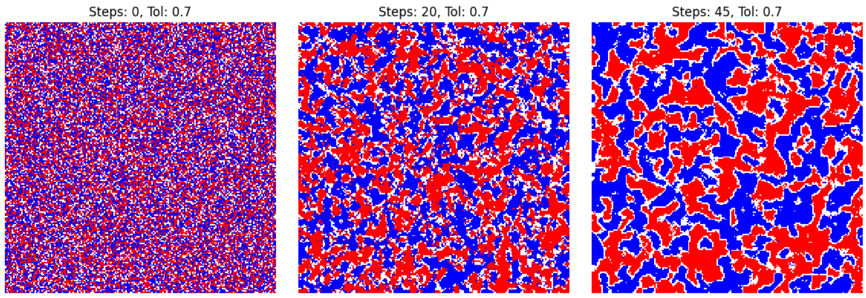

Figure 1: Process, with increasing step count

In the first grid seen in the figure, there is a state where no steps have been taken (original state). In the second grid, the shape formed with 20 steps is indicated, and in the third grid, with 45 steps, as can be clearly seen, the separation becomes more clearly apparent as the number of steps increases. 1

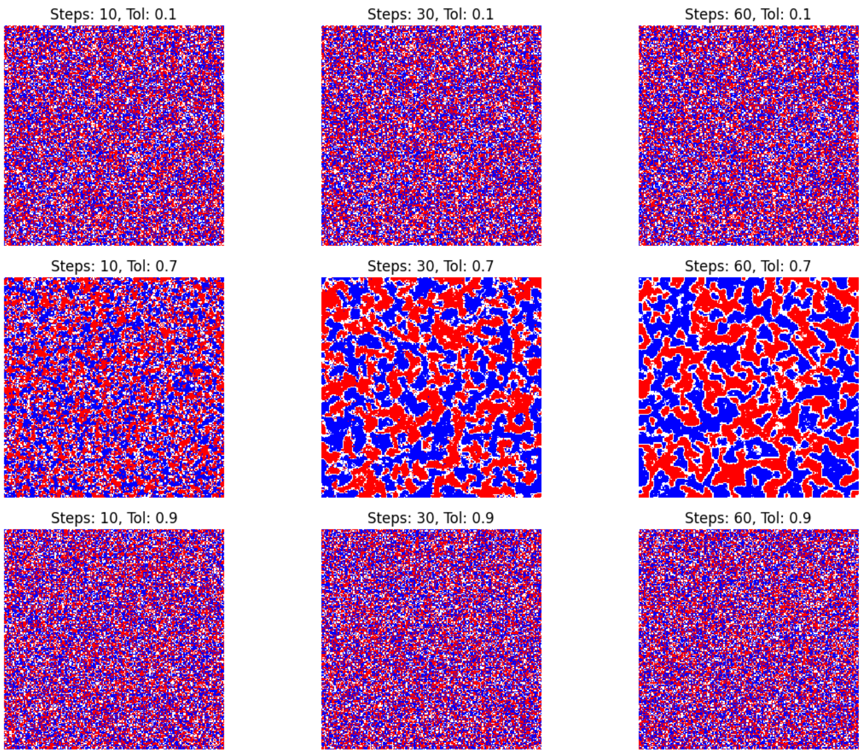

Figure 2: Some grids, by tolerance and step size

What the tolerance value indicates can actually be called the agent's condition of being happy, if it is 0.1 the agent tends to be happy no matter what happens around him, if it is 0.9 it will be difficult to provide the agent's condition of being happy. In the examples where I set the tolerance to 0.1, no segregation is seen, because the agent is already happy with the situation he is in, does not feel the need to have his own race around him and does not move. When I set the tolerance to 0.9, the grid still does not appear segregated, because agents want all 8 sides to be filled with their own race, and since it is difficult to provide such a situation, they cannot reach a stable state. As can be seen in the figure 2 above, the best segregations occur when the tolerance is 0.7.

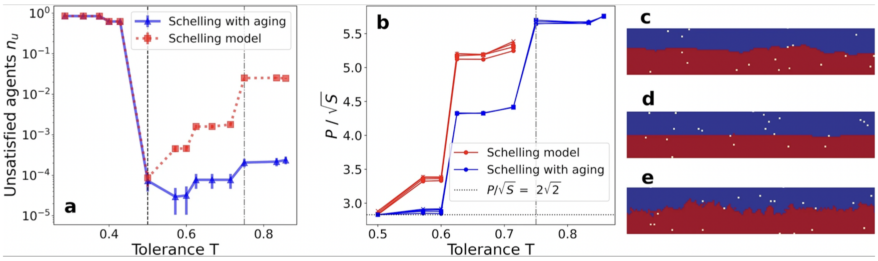

Figure 3: Tolerance values

I took the above graph from an article called Aging effects in Schelling segregation model [2], a topic discussed in more detail.

2 Code

There were 3 different file in the assignment; the first was a segregation model written in Julia, the second was an empty Haskell file, and the last was a Julia code that would run this empty Haskell file. Some functions and parameters were given in the Julia code, but I couldn't figure out how to use them while running Haskell, so I wrote everything from scratch in Haskell.

```haskell
-- Parameters
gridSize :: Int
gridSize = 200

tolerance :: Float
tolerance = 0.7

steps :: Int
steps = 5
```

Since the variables already defined in the Julia code that will run the Haskell code were not given in the run command, I could not access them, so I preferred to simply check the variables like above.

2.1 2D Grid

First of all, I wanted to convert the txt given in the assignment into a 2D integer array, this would make my job much easier when checking the neighbors.

```haskell
-- Type definitions for 2D grid
type Grid = [[Int]]
type Position = (Int, Int)
```

I created two different types as above, Grid for 2D array and Position to access its indexes.

```haskell
-- Load grid from file
loadGrid :: FilePath -> IO Grid
loadGrid filename = do
    content <- readFile filename
    let values = map read (words content) :: [Int]
    return (chunksOf gridSize values)
  where
    chunksOf _ [] = []
    chunksOf n xs = take n xs : chunksOf n (drop n xs)
```

The above function reads the file from FilePath, splits them by spaces, converts them to integer and puts them into the values variable as an array. Then, it creates a 2D array, namely a grid, by separating it into chunks. Since it is not a pure function (it may produce different results for the same input, the content of the txt file may change), it is necessary to use the IO monad.

```haskell
-- Get the element at a position in the grid
getCell :: Grid -> Position -> Int
getCell grid (x, y) = (grid !! x) !! y

-- Set the element at a position in the grid
setCell :: Grid -> Position -> Int -> Grid
setCell grid (x, y) value =
    let row = grid !! x
        newRow = take y row ++ [value] ++ drop (y+1) row
    in take x grid ++ [newRow] ++ drop (x+1) grid
```

Then, for arranging and accessing to this grid, the getter and setter functions come. The getter function works very simply, in Haskell, the `!!` operator allows me to access the index and thanks to this, I can first select the row and then the column and go to the address I want. The setter is a bit more complicated, first I give the function an address and a value to be set, then the row at the given position is found, values are taken up to the column to be added on this row, the value to be added is placed, and the remaining values are placed with `drop`.

```haskell
-- Save grid to file
saveGrid :: FilePath -> Grid -> IO ()
saveGrid filename grid = do
    let flatGrid = concat grid
    writeFile filename (unlines (map show flatGrid))
    putStrLn ("Simulation finished. Output saved to " ++ filename)
```

I need a saver function to save this grid in an output file at the end of the program. This function first converts the grid to a 1D `Int` array with `concat`, then converts integers to strings with `map show`, joins these strings with `"\n"` with `unlines`, and finally prints to output file.

2.2 Randomness

Randomness has a very important place in the segregation algorithm. Thanks to randomness, results closer to real life are produced, if there is no randomness, a priority order will emerge between positions, which can negatively affect the algorithm.

```haskell
-- Simple RNG function, GCC version
nextRandom :: Int -> Int
nextRandom seed = mod (1103515245 * seed + 12345) 2147483648

-- Generate a random number between min and max
randomRange :: Int -> Int -> Int -> (Int, Int)
randomRange minVal maxVal seed =
    let newSeed = nextRandom seed
        range = maxVal - minVal + 1
        value = minVal + (mod newSeed range)
    in (value, newSeed)
```

I generate a random number within a range using the functions above. The numbers in the `nextRandom` function are the numbers used in GCC's random number generation functions [3]. In the `randomRange` function, I simply take `minVal` as the base and add a value that will not exceed the range.

```haskell
-- Fisher-Yates shuffle algorithm with simple RNG
shuffle :: [a] -> Int -> ([a], Int)
shuffle xs seed =
    shuffleHelper xs [] seed
    where
        shuffleHelper [] acc seed = (acc, seed)
        shuffleHelper xs acc seed =
            let len = length xs
                (idx, newSeed) = randomRange 0 (len-1) seed
                x = xs !! idx
                rest = take idx xs ++ drop (idx+1) xs
            in shuffleHelper rest (x:acc) newSeed
```

Randomness will come in handy when shuffling an array. The algorithm above creates an extra array to shuffle an array and randomly selects an item from the main array and adds it to the beginning of the new array, then deletes the item from the main array. This process continues until the main array is empty.

2.3 Segregation Logic

```haskell
-- Run simulation
runSimulation :: Grid -> Int -> Int -> (Grid, Int)
runSimulation grid seed 0 = (grid, seed)
runSimulation grid seed n =
    let (grid', seed') = schellingStep grid seed
    in runSimulation grid' seed' (n-1)
```

The above code allows `shellingStep` to run as many times as desired. The new step is done on the changed grid, and the new seed randomness also increases its relevance to real life.

```haskell
-- One step
schellingStep :: Grid -> Int -> (Grid, Int)
schellingStep grid seed =
    let (unhappyAgents, emptyCells) = findUnhappyAndEmpty grid
        (shuffledUnhappy, seed') = shuffle unhappyAgents seed
        (shuffledEmpty, seed'') = shuffle emptyCells seed'
        moves = min (length shuffledUnhappy) (length shuffledEmpty)
        movePairs = zip (take moves shuffledUnhappy) (take moves shuffledEmpty)
    in (foldl' moveAgent grid movePairs, seed'')
```

The `schellingStep` function first finds the `unhappyAgents` and `emptyCells`. Then it shuffles them with different seeds. The moves variable holds the size of the shortest list, and the `movePairs` variable zips these two lists together. For example, (`unhappyAgent1`, `emptyCell1`). Finally, the `moveAgent` function works with `foldl'` and moves all agents to empty positions.

Usually the choice is between `foldr` and `foldl'`, since `foldl` and `foldl'` are the same except for their strictness properties, so if both return a result, it must be the same. `foldl'`is the more efficient way to arrive at that result because it doesn't build a huge thunk [4]. Since the program works on large arrays, `foldl'` is a more suitable choice.

```
-- Move an agent to an empty position
moveAgent :: Grid -> (Position, Position) -> Grid
moveAgent grid (unhappyPos, emptyPos) =
    let agent = getCell grid unhappyPos
        grid' = setCell grid unhappyPos 0
    in setCell grid' emptyPos agent
```

The `moveAgent` function simply takes `unhappyAgent`, makes its current location empty, and moves it to `emptyPos`.

```
-- Find all unhappy agents and empty cells
findUnhappyAndEmpty :: Grid -> ([Position], [Position])
findUnhappyAndEmpty grid =
    let positions = [(x, y) | x <- [0..gridSize-1], y <- [0..gridSize-1]]
        isEmpty pos = getCell grid pos == 0
        isUnhappyAgent pos =
            let v = getCell grid pos
            in (v /= 0) && not (isHappy grid pos)
        emptyCells = filter isEmpty positions
        unhappyAgents = filter isUnhappyAgent positions
    in (unhappyAgents, emptyCells)
```

The `findUnhappyAndEmpty` function is actually a function that goes through the entire grid and collects `emptyCells` and `unhappyAgents`. It collects these values by producing 2 different conditions `isEmpty` and `isUnhappyAgent`. If the value in the cell is 0, it is `emptyCell`, if the value is not 0 and the agent is not happy (controlled by `isHappy` function), it is `unhappyAgent`. The `/=` operator is equivalent to `!=` in other languages.

```haskell
-- Check if agent is happy
isHappy :: Grid -> Position -> Bool
isHappy grid pos =
    let agent = getCell grid pos
    in if agent == 0
       then True
       else let neighbors = getNeighbors grid pos
                similar = length (filter (== agent) neighbors)
                total = length (filter (/= 0) neighbors)
            in if total == 0
               then False -- all of the neighbors are different from agent
               else (fromIntegral similar / fromIntegral total) >= tolerance
```

The `isHappy` function determines whether the agent in a position is happy. If the position is empty (0) it returns happy. If not, it checks the values of all neighbors with the help of the `getNeighbors` function and creates similar and total values. If the ratio of these values is greater than the tolerance, the agent is unhappy and returns False, otherwise it returns True and the agent is happy.

```haskell
-- Get neighbors for a cell
getNeighbors :: Grid -> Position -> [Int]
getNeighbors grid (x, y) = do
    dx <- [-1, 0, 1]
    dy <- [-1, 0, 1]
    if dx == 0 && dy == 0
        then []
        else
            let nx = x + dx
                ny = y + dy
            in if 0 <= nx && nx < gridSize && 0 <= ny && ny < gridSize
                then return (getCell grid (nx, ny))
                else []
```

The `getNeighbors` function is used to check the 8 neighbors around an agent and returns these values as an array.

REFERENCES

[1] Wikipedia contributors. Schelling's model of segregation. https://en.wikipedia.org/wiki/Schelling%27s_model_of_segregation, 2024. Accessed: 2025-05-09.

[2] D. Abella, M. San Miguel, and J. J. Ramasco. Aging effects in schelling segregation model. *Scientific Reports*, 12:19376, 2022.

[3] Wikipedia contributors. Linear congruential generator — Wikipedia, the free encyclopedia, 2025. [Online; accessed 9-May-2025].

[4] HaskellWiki. Foldr foldl foldl' — haskellwiki,. https://wiki.haskell.org/index.php?title=Foldr_Foldl_Foldl%27&oldid=62842, 2019. [Online; accessed 9-May-2025].