BLG 322E - Computer Architecture

Assignment 2

Feyza Nur Keleş
820220332

May 12, 2025

Contents

1 Problem Definition

1.1 What is The Schelling Segregation Model?

The Schelling Segregation Model is an example of agent-based modeling used to study patterns of social segregation. In this model, individuals (agents) of two types are placed on a grid and moved based on their satisfaction based on their local neighborhood criteria. Even when individual preferences for similar neighbors are very mild, the result is a very segregated pattern.

1.2 Objectives

In this assignment, we were given 2 julia files, one for running the simulation and one to provide an example with a skeleton that suggests an intermediate approach with only one ‘Agent‘ class and the type of agent encoded as an integer, where 0 represents an empty cell. We were expected to write an algorithm in haskell, and we were free to use the given algorithm or write one from scratch. Mine wasn't so different but I didnt use an agent class directly.

2 Implementation Motivation

I used the provided Julia code to understand the logic behind the Schelling model, especially how to check if an agent is happy and how the movement works. At first, I tried moving unhappy agents from left to right in the grid, but that caused some issues with the results (explained more in Section **??**). Later, I changed it and got better results. Other than that, I mostly followed the structure of the algorithm and focused on getting it to work correctly.

3 Implementation Explanation

3.1 Overall Algorithm

The simulation starts with putting the agents into a grid. At each step, it checks if agents are happy based on who's around them. If they're not happy, they move somewhere else. This loop runs for a number of steps. At the end we see the grid with `heatmap` to show how things turned out.

3.2 Haskell Implementation

This section explains my Haskell code for the Schelling model simulation. I read the grid from a list provided by simulation jl files, and write the final grid back to a list and return it.

Basic setup First, i define some constants:

```
grid_x = 200
grid_y = 200
threshold = 0.4
radius_neighbour = 5
maxSteps = 1000
```

The grid is 200x200, and agents are happy if at least 40% of their neighbors are same type as them. Radius is 5, so each agent looks around a big area.

Grid conversion I read the input from a file as a flat list, and turn it into a 2D grid with this:

```
toGrid xs = [ take grid_y (drop (i * grid_y) xs) |
i <- [0 .. grid_x - 1] ]
```

Flattening back to a list is just `concat` basically.

Random shuffle First, I tried doing without randomness when trying to find a happy spot, but bottom to up iteration caused a faulty output like this figure below.

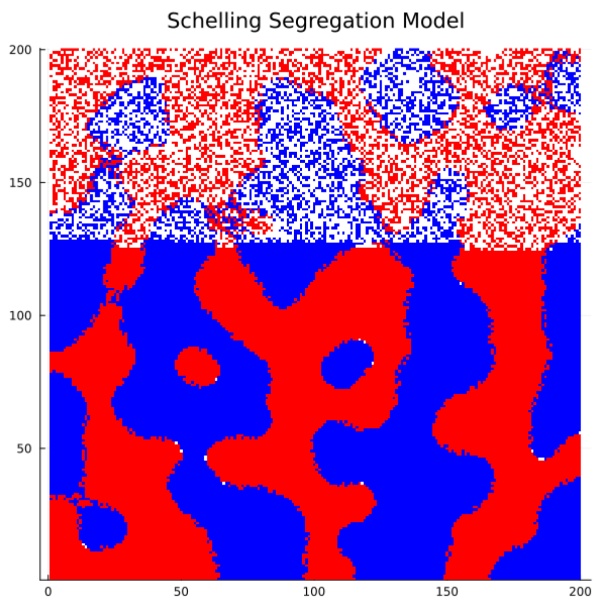


Figure 1: Faulty Output

I couldn't install the random library honestly. So I tried using time to generate a kind of randomness. I used this to shuffle the empty grids to search from.

```
shuffle xs = do
  t <- getPOSIXTime
  let seed0 = floor (t * 1000)
  return $ shuffle' seed0 xs
```

Neighbourhood and Happiness Each agent looks around and checks its neighbours:

```
neighbors grid i j = ...
isHappyAt grid i j t = ...
```

If there are no neighbors, it's counted as happy. If there are some, it checks how many are same type. If enough percent (like more than threshold), then it's happy.

One timestep In one step, I shuffle all coords and empties, then go over all cells:

```
timeStepIO grid = do
  coords  <- shuffle [...]
  empties <- shuffle [...]
  return $ go grid empties coords
```

In the loop, if an agent is unhappy, it tries to move to the first empty spot where it would be happy. If no spot found, it stays. If found, it swaps places. I use `replaceCell` twice to do that.

Termination Condition The simulate function runs until all agents are happy or we reach maxSteps(which is 1000):

```
simulateIO :: Grid -> Int -> IO Grid
simulateIO grid stepCount = ...
```

Main function In the main function I read the file, turn it to grid, run the simulation, and write the result:

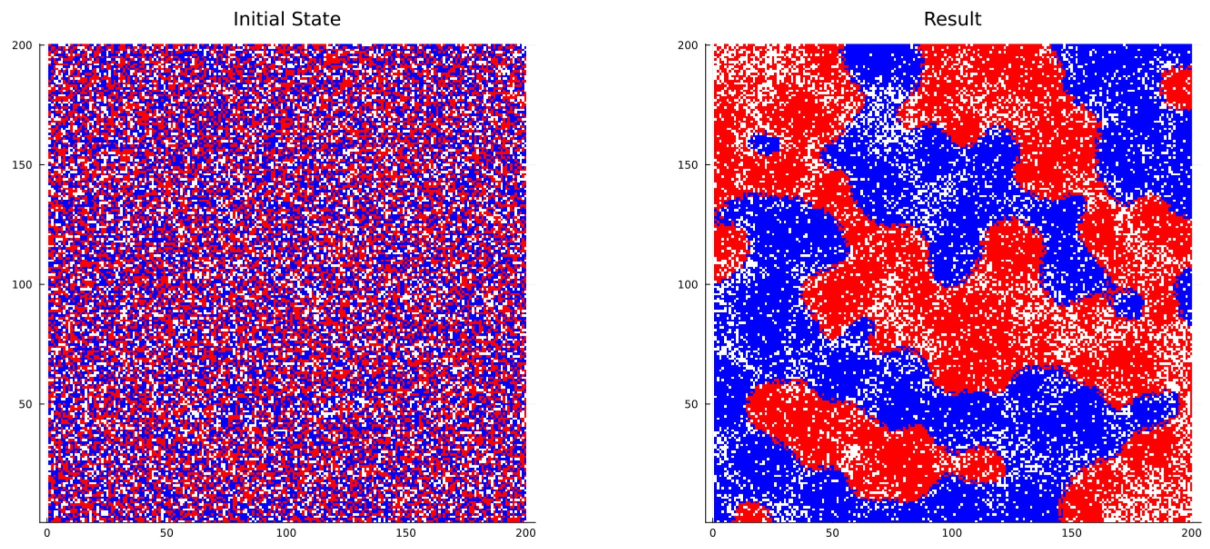
```
main = do
  args <- getArgs
  case args of
    [inputFile, outputFile] -> do
      contents <- readFile inputFile
      ...
```

The filenames are passed as arguments. If there's not 2 args, it just prints a usage message.

Summary The movement logic is correct and the shuffle makes the simulation fair. I had some trouble with random numbers but the POSIX workaround worked great. It's slower than maybe other implementations, since I iterate for shuffling too many times, but haskell itself is very clear and pure code mostly.

4 Results

Initial vs Final State. Below u can see two side-by-side images: one at the very start of the simulation, and one after it finishes. At first, all the agents are shuffled randomly on the grid. After some iterations, they start moving to spots where they're happier, and we get this clear separation by type.

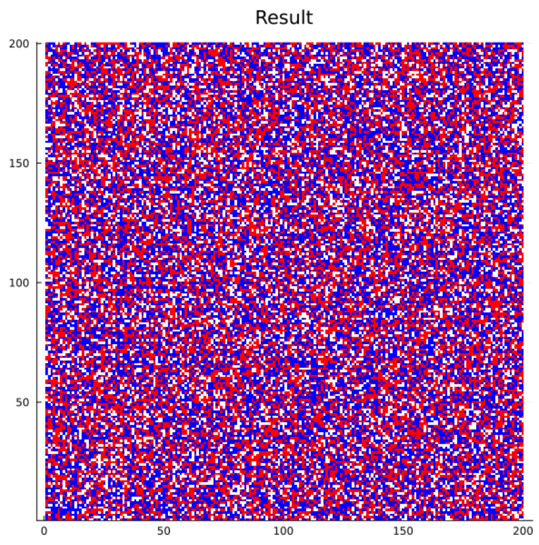


(a) Initial grid (input) (b) Final state

Figure 2: Comparison of initial and final states of the grid with happyness %1.0

Segregation Behavior Even though the threshold was just `0.4` (so agents are ok with only 40% similar neighbours), they still ended up forming visible clusters. It's interesting how small preferences like this can cause such strong separation just from local rules.

Parameter Sensitivity To better understand the effects, i tried a few variations on threshold and the grid with threshold 0.3 and 0.2 looks more mixed because agents are more tolerant and don't move much. Even though it ends with 97.7% happiness, it's much less clustered. At threshold 0.5, agents are pickier and move more, allowing to reach 100% happyness. This causes sharper red and blue regions, with white (empty) cells often forming buffer zones between groups. So even though both cases reach high happiness, the social structure is very different.



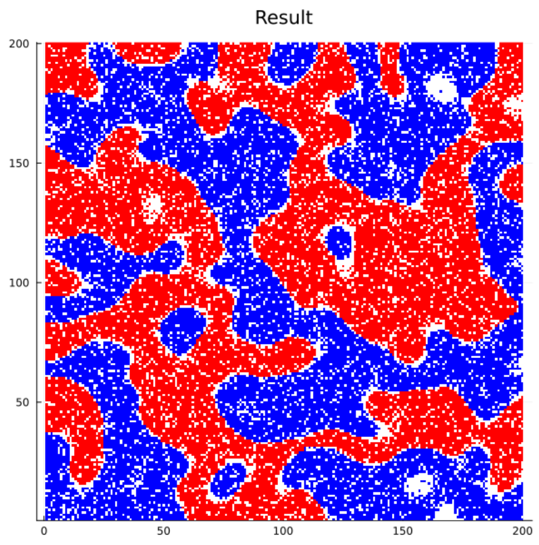


Figure 3: Left: Threshold = 0.3 (tolerant). Right: Threshold = 0.5 (strict).

5 Conclusion

From the visual results and after playing with different parameters, i noticed that even when the agents have low similarity requirements (like a small threshold), they still tend to form visible clusters. This means the model shows how even simple, local preferences can lead to big global patterns. When the threshold is increased too much tho, sometimes the agents can't find enough neighbors they like, so many of them stay unhappy or don't move. It also depends a lot on the neighborhood radius—larger radius makes agents consider more neighbors, which can make the model slower to settle or make it more likely to mix.

I learned that the combination of radius and threshold really affect how the groups form or if they stay mixed. For example:

- Low threshold → tolerant agents, but still cluster
- High threshold → can't find good spot, stuck
- Bigger radius → more global info, slower but maybe more fair result (took too much time since I didn't have an efficient random function so I didn't include any output figures)

One other thing I observed was when I tried no shuffle, the patterns were more clustered and white areas werent spread accross the grid. So randomness helps avoid those edge-case outcomes.

Overall, an efficient simulation was achieved through careful algorithm design and problem-specific optimizations and solutions. The process showed the importance of performance considerations and creative workarounds when faced problems in environment. And this algorithm showed that even the slightest change in the constraints can lead to huge segregations and clustering.