Math 173: Final Exam

Ruben Arenas

December 12, 2014

Due by December 18, 11:59 PM, in your DropBox
30 points possible

1 Modeling Housing Bias

In this exam we will explore how individual bias leads to macro level patterns. Assume there are two types of people – red people and green people. Although these people have no problems with each other they do prefer to live near their own kind. We call this inclination "bias". Bias is a percentage that represents the amount of your own kind you need surrounding you before you become unhappy. If your bias is 30% then you would want 30% of your neighbors to be like you, or else you would be unhappy.

In our model we have a grid system that represents places that individuals can live. A neighborhood around a person is the squares immediately adjacent to the person. For example, suppose that both red and green people have a bias of 1/3, so that these people would want 1/3 of people in their neighborhood to be like them. In such a situation, the example in Figure 1 could occur. The central resident has 4 neighbors, only one of which

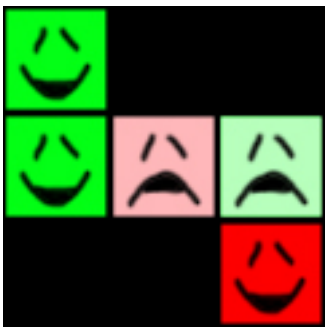

Figure 1: A hypothetical neighborhood with 1/3 bias.

is like it, giving a similarity of 25%. Since this is less than 1/3 the central resident is unhappy. The lower right resident has 2 neighbors, one of which is like it. In this case there is 50% similarity which exceeds the bias so this resident is happy.

Residents of our fictional world have 3 states:

- Happy – this occurs when the bias is met or exceeded *and* there is at least one neighbor of the opposite type. Our residents enjoy at least a some diversity.
- Indifferent – this occurs when the bias is met or exceeded *and* there are no neighbors of the opposite type.
- Sad – this occurs when the bias is not met.

The central hypothesis of this exam is that unhappy residents will move in order to become happy, or at least indifferent. A reordering of the example above to make everyone at least indifferent is shown in Figure 2.

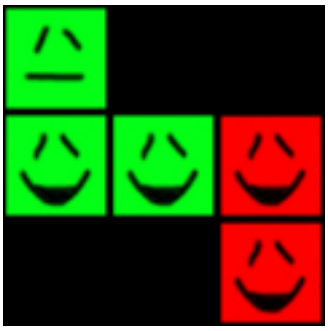

Figure 2: A reordering of the previous neighborhood to satisfy everyone.

2 Measuring Segregation

The pattern in which people live in proximity with people from their own group is called "segregation". We will create a metric to measure segregation.

First, we begin by computing "local similarity" – a measure of how much segregation there is in a neighborhood around a given resident (we ignore anyone outside of the neighborhood). In our model, it is computed as follows

$$\left| \frac{\text{red people}}{\text{people in neighborhood}} - \frac{\text{green people}}{\text{people in neighborhood}} \right|.$$

In Figure 1, the central resident has a local similarity of

$$\left| \frac{2}{5} - \frac{3}{5} \right| = \frac{1}{5},$$

or 20%. Local similarity ranges from 0%, complete equality of representation, to 100%, complete segregation. Verify for yourself that the upper-left resident has a local similarity of 33%.

Note that local similarity is not computed if there is no resident at the center of a neighborhood.

Finally, segregation is defined as the average of all local similarities, that is, the sum of all local similarities divided by the total number of residents. The segregation in Figure 1 is roughly 30.7%, while the segregation in Figure 2 is about 57.3%.

3 Exam Part 1: The Exploration Tool

In this part of the exam you will create a tool where users can explore the effect of bias on housing patterns. You may model this portion after "Exploration mode.exe" (make sure all bitmap files are in the same directory as the executable before you test it).

Your program should begin by querying the user for

- The bias, expressed as a decimal number
- The proportion of red residents, a decimal number between 0 and 1 inclusive.
- The proportion of green residents, a decimal number between 0 and 1 inclusive.

The two proportion values do not need to add up to 1. There can be empty space (unoccupied houses).

The program should generate a 15 unit by 15 unit grid of randomly placed red and green residents in the proportions specified. These residents should either be happy, sad, or indifferent according to the rules described above.

At this point the user is allowed to move residents using the mouse to unoccupied positions. The user may only move residents who are sad – indifferent and unhappy residents can not be moved. Keep in mind that by moving residents the state of neighboring residents will tend to change – someone who was happy before may become sad.

The program must report the level of segregation in real-time as the user moves around residents. When all residents are happy or indifferent the user can no longer interact with the grid.

Extra credit: Pressing 'r' reloads the grid in a new configuration. This is *not* easy! Your understanding of memory management will need to be perfect.

4 Exam Part 2: The Simulation Tool

In this portion of the exam you will write a program that will start with a random housing configuration and move around pieces in an effort to make everyone content. You may model this portion after "Simulation mode.exe".

The program should work like exploration tool except the user can not interact with the generated map via the mouse. Rather, when the user presses 's' a simulation should start. This simulation *randomly* selects unhappy residents and moves them into *random* empty spots. Keep in mind that by moving the resident in question you may make others

unhappy. The program may run forever depending on the level of bias. The segregation should be reported in real-time.

The naive code for this is straightforward, but your program will run very slow, especially as it nears completion.

Extra credit: Get your code to run as fast as my example for bias levels less than 70%.

5 Advice

Here are some general thoughts to make your life easier

1. The more time you spend on design early on, the easier this will be to code. This program is straightforward with a clear object-oriented design.
2. With the above in mind, start by identifying all the necessary objects as well as what they should do, and what properties they should have. Spend time answering difficult questions – should the red and green pieces be represented by separate classes? How will you keep track of where everything is?
3. How should your objects interact? Don't ever let a class do something it shouldn't be able to do according to its nature. For example, a single resident object should not be able to know where the other residents are without help from the grid object.
4. One object you definitely want is a well-designed Image class. An Image represents a graphic read from a file (I provided you with my bitmap files). An Image should be able to load graphic data from a file (use readimage, imagesize, getimage) and draw that graphic to the screen (use putimage). You definitely want to use dynamic memory here, so make sure you have everything you need!
5. Keep in mind that if you are running your code from Visual Studio your image files (or mine in this case) need to be in the same directory as your source code.
6. You will want some sort of class or classes representing the residents. You can load the three Image's directly into the class at construction time and then cycle through them depending on the state of the resident.
7. You will want a grid class of some sort that stores residents. Think Tetris. With that said, in this case it is much easier to have a 2D array of resident pointers instead of the actual objects themselves! As long as you can wrap your head around the pointers this will simplify things. If you can't wrap your head around this you'll need to find another route.
8. The classes I used in the exploration tool are identical to the ones in the simulation tool. The only thing that changed was my code in main using those classes. Now that's good design.
9. Mouse input can be very difficult. Avoid using getmouseclick with WM_MOUSEMOVE. Use mousex and mousey instead to get the mouse coordinates. To detect a press

or depress use code similar to our keyboard capture code – and in this case getmouseclick works fine.