# Documented design

Because my project is made of so many parts I decided to split my work into stages, working on one stage at a time and keeping them separate until they need to be put together. Within these stages I will be working on smaller ‘sections’ of them to break it down even more. This way I can focus on one stage of the programming until it works completely then move on, not having to worry about any previously completed stages, and the same goes for each section.

## Stage 1-

**The main simulation part of the program, most similar to Schelling’s model.**

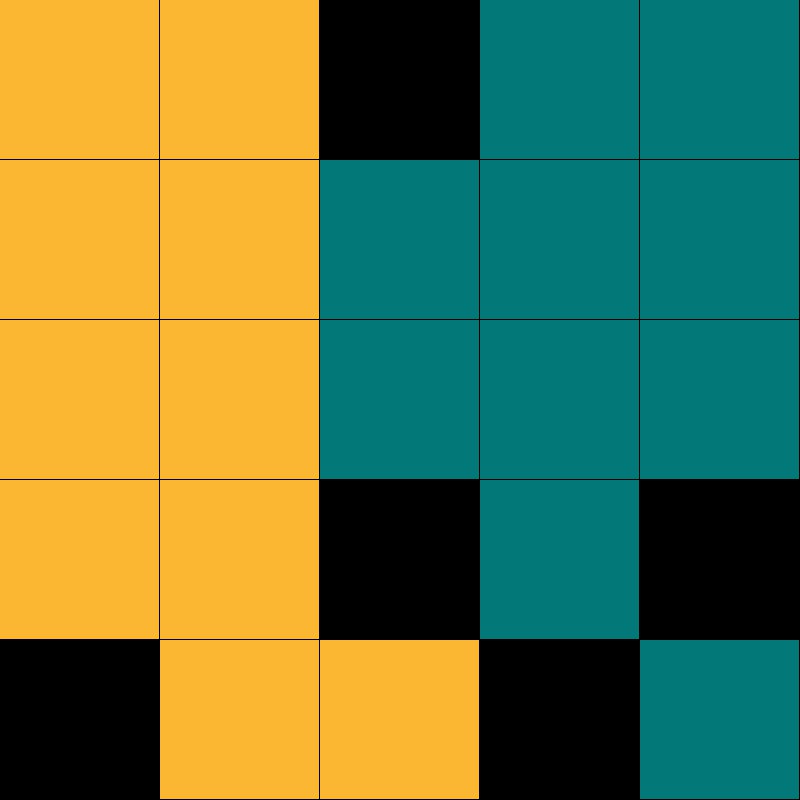
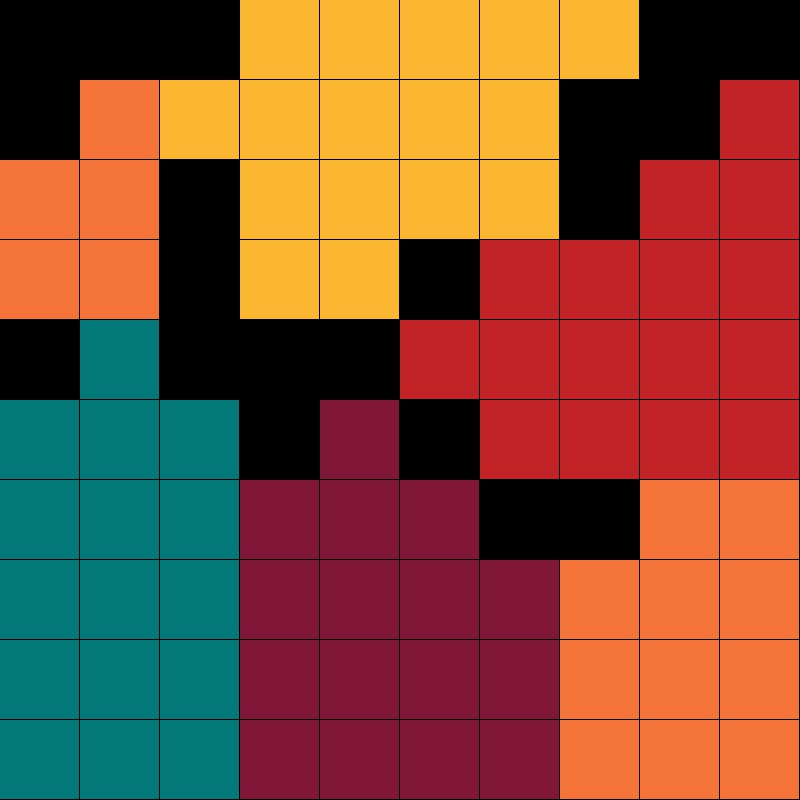
### Section 1

The basic code of the movement and generation got the board. For now, there will be no user-set variables or pygame use, instead focusing only on the actual functional part of the code.

### Section 2

In this section I will allow the user to input their own values for the variables that affect the simulation, such as tolerance factor, diversity factor, how many groups etc. The tolerance factor is the percentage of adjacent neighbours each agent has that have to be like them at a minimum in order for that agent to be happy, whereas the diversity factor is the minimum percentage of neighbours of a different group each agent wants in order for them to be happy.

### Section 3

Using pygame for the first time to create the UI for this stage of the simulation. See ‘Drawing the table’ in ‘Description of algorithms’ for how I’m going to do this. Here are some examples:

## Stage 2-

**Dijkstra’s algorithm**

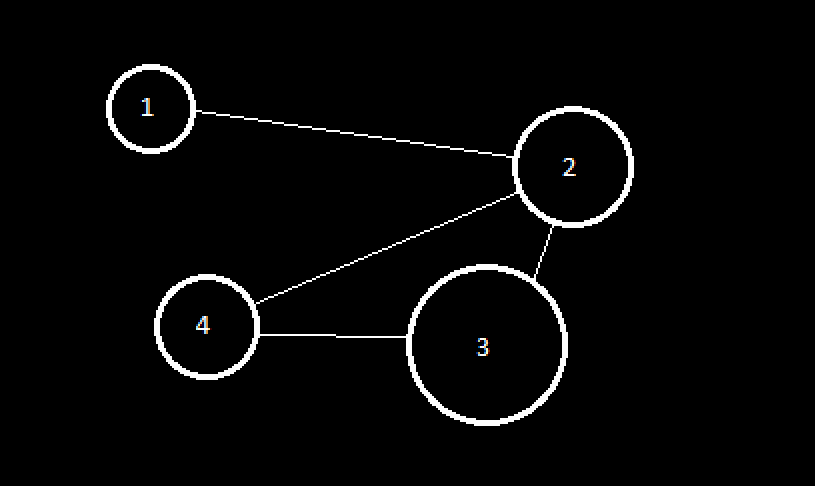
### Section 1

The next section of code I will be working on is Dijkstra’s algorithm in a separate program without pygame. I will be using a separate program as until Dijkstra’s algorithm works completely it would only make it more difficult to implement if I were to work on it within the main program. For the first section of this stage I will be making a class called ‘nodes’ and randomly generating the positions of a set number of instances (nodes) and writing an in-class function that will retrieve the length between the instance that ran the function and any other node. Each node will also have an individual ‘size’, which I will explain more on later.

### Section 2

Next I will be writing an in-class function called ‘getAdjacent’ (or something similar) which will make a list of the of all the other nodes that are in range of the initial node. These nodes are connected by ‘edges’, and if all nodes are not connected in some way they will re-generate until they are. I will also be giving each node a unique iterating ID.

### Section 3

Now that I have coded all the basic functions for the nodes I will use pygame to put them in a UI, with edges between each node based on the list of adjacent nodes, with the edge length also on the edge and the node’s ID inside of the node, which will be represented by a circle of radius proportional to their ‘size’. This is an example:

### Section 4

I will next give the user the option to choose how many nodes there are and pick a starting and ending node, and implement Dijkstra’s algorithm in this section. See ‘Dijkstra’s algorithm’ in ‘Description of algorithms’ for how I’m going to do this. The user will be able to select nods by clicking on them, which will make them change colour slightly. This is for a later part of the program.

## Stage 3-

**Making the live unhappiness graph**

### Section 1

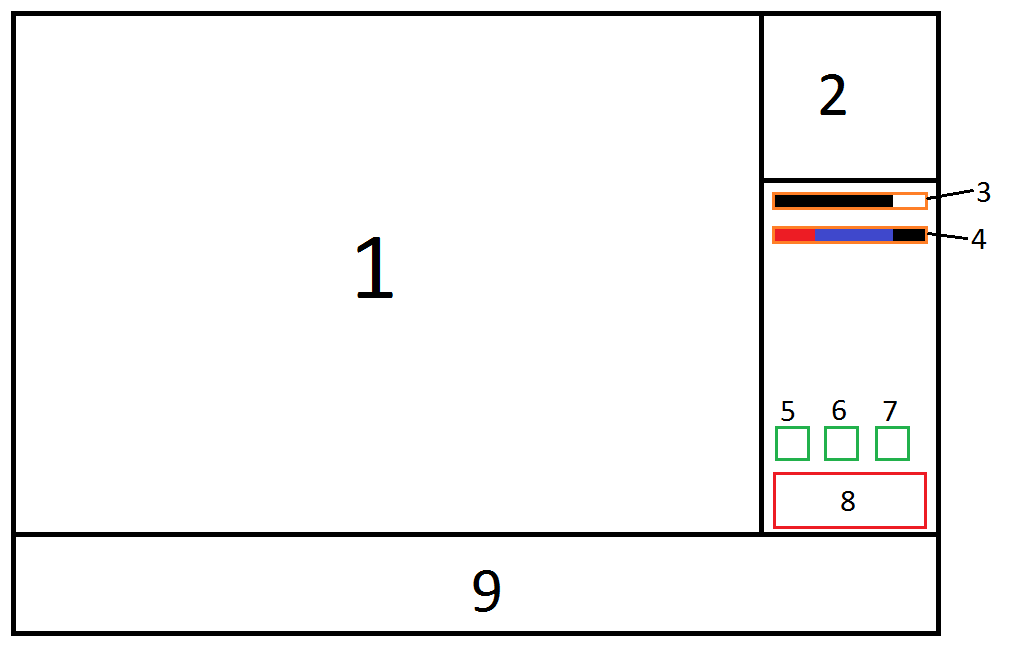
I will use [matplotlib] to make the axis for the graph in a window, as the graph itself will need to be in the same window as everything else in the simulation. For now, the graph won’t be made live, but will instead display any data I feed into it manually as a test.

### Section 2

In this section, I will work on making the graph live, displaying a set of random data. After this is complete, the only difference between this graph and the graph that will be in the real thing is the data fed into it, so it is essentially done.

## Stage 4-

**Making the template and pause button. End goal:**

****

### Section 1

As a first step, I will be making the basic look of the template in pygame, which is only the black outline in the above diagram. Section 1 will show the board of the selected node, 2 is where the node map will be, 3 is the average segregation of the current node and 4 is the proportion of the groups and blanks. 5, 6 and 7 are the pause, slow motion and play buttons respectively. 8 is the exit button, which will exit from the simulation back to the main menu. 9 is the section where the live graph showing the total average unhappiness across all nodes is.

### Section 2

After making the basic outline I will be making a ‘button’ class which will have its position and click function etc. (More on the button class in the class definitions table) Instances of the button class will be made for 5, 6, 7 and 8. Because there is nothing to affect, none of them with the exception of the pause button will have an effect. I can test the pause button by making a main loop in the code (which is where all the other code from other stages will be) and having it constantly print out something arbitrary, then stop when the pause button is pressed and resume when it is pressed again.

## Stage 5-

**Making the user selection screen**

### Section 1

This entire stage will involve no pygame as it would make it unnecessarily complex. In section 1 I will be letting the user input all the changeable variables with no error checking. The variables they can set are:

* How many nodes
* If they want each node’s bias and diversity factors to be the same
* Bias and tolerance factors
* The size of each node
* The number of groups
* The exact number of each group/how many blanks in each node

### Section 2

In this section, I will be adding error checking and limitation checking- that is, because there are some variables that affect each other, I will have to limit them in relation to each other. For example, the bias and tolerance factors cannot add up to more than 100 and the number of agents and blanks cannot exceed the size of each node squared.

## Stage 6-

**Putting it all together**

Because all I need to do in this stage is put all the separate parts of my program together there are no sections. After this stage, my entire program should be finished

## Data structure table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Purpose | Type | Size | Example |
| colours | Contains the RGB values for the colours of each group and their corresponding group ID. | Dictionary | The maximum number of groups, which is 5. | {0:(118,54,38), 1:(144,175,197), 2:(51,107,135), 3:(42,49,50),  4:(0, 150, 150)} |

## Description of algorithms

### Dijkstra’s algorithm:

This algorithm finds the shortest path between two nodes in a map[[1]](#footnote-1) along its edges. It works by getting the starting node and fetching the lengths of all the connected edges, as well as the nodes they lead to, and adding them to a list. The shortest edge is then selected from the list and the node it leads to is added to a dictionary of nodes and the **cumulative edge length** to get to that node, containing that node and the starting node. All of the new node’s edges are added to the list of edges.

Next, the shortest edge from that list is selected and the node it leads to is the next node (and that edge is removed from the list), whose edges are then added and it goes on until the end node is reached. Then starting from the end node you work backwards, and if the difference in two node’s cumulative lengths is equal to the edge length between them that node is the next node. You reach the start node again by working backwards like this.

### Drawing the table:

Drawing the table is a fairly complex procedure as while the size of my screen is a fixed value the number of spaces is not, and all spaces must be the same size and fill up the screen entirely. To overcome this problem, I need to calculate the ratio of the size of the board to the number of spaces on one side as there can only be a square number of spaces. For example, if the user chose for the board to be of size 30, there would be 900 (30x30) spaces total and 30 is the value that would be used in the calculation. For example, if the screen was 1000x1000 pixels and the user chose a size of 30 the calculation would be:

K = 1000/30, where K is the number of pixels each space needs to be across to fit perfectly onto the board.

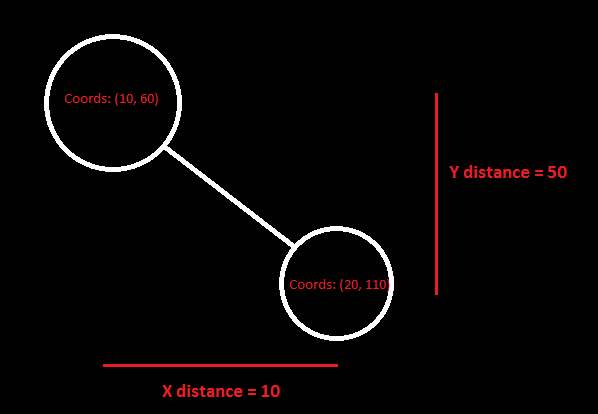
Then, in order to draw the board itself, I need to enumerate through the table containing the layout of the board to get the rows of the board, then enumerate through those to get the individual agents as well as their positions in the board, for example the top left space would be at position (0, 0) and the one below it would be at (0, 1) etc.

To draw the spaces onto the board I multiply their x positions and y positions in the board by K to get their position on the screen, then make their width and height K-1, so they fit together perfectly with small gap between each of them for aesthetic purposes.

### Moving the agents between nodes:

This algorithm involves checking how many times an agent has moved and been unhappy in a row by iterating a value in that class for the instance (agent) in question whenever it moves and is unhappy and setting it back to zero when it moves and is happy. If the agent moves and the value is iterated (agent is unhappy with new position) and the value goes over a certain threshold, which will be the same for each agent, a different node in the map will be randomly selected for that agent to move to.

A path to that node from the current node the agent is in will then be generated using Dijkstra’s algorithm and the agent will begin its journey to that node. Every time the agent makes a turn I will have to calculate its movement from its current coordinate to its destination. I will do this with an in-class function called ‘getDir’ which will calculate exactly how many pixels in the y direction and how many pixels in the x direction the agent will have to move each tick to get to the next node. This can be calculated by getting the Y distance between the two nodes and dividing it by some constant which will be all agent’s speed then doing the same for the X distance. For example:



Say the agent is moving from the top left node to the bottom right node with a speed of 5. The Y distance it needs to move each tick is 50/5, so 10. The X distance would be 10/5, so 2. So every tick the agent would move 2 pixels across and 10 pixels down (as in python the farther down you go the larger the Y value gets). I will reach its destination in 5 ticks.

Once the agent reaches the end node in its path (which will be checked by getting its coordinates every tick and comparing them to the coordinates of the destination node) it will disappear from the screen and be put into a random blank space in that node.

If there are no blank spaces the agent will move to another random node. If there is an empty space, then the in-class variable ‘currentNode’ will be changed to the node the agent has moved to.

## Description of the simulation process

There are several ways I could implement the main simulation and I have described each in the following table:

|  |  |  |
| --- | --- | --- |
| Description | Advantages | Disadvantages |
| Select random agents until an unhappy one is found, then move it | Easy to code, takes very little code | Might take a long time to find unhappy agents if few are left, no way of knowing if it’s done or not without checking the entire grid after each and every change, would take a long time especially if the unhappy agents aren’t selected for a long time. Overall unreliable |
| Cycle through each agent from top to bottom and moving the unhappy ones | Easy to code, able to tell when all agents are happy | Looks bad, might affect the end result due to the nature of this implementation. Forced to check every agent even if they’re happy |
| Make a list of unhappy agents at the start, then modify it as the simulation takes place | Efficient, easily able to tell when the simulation is finished immediately | Takes the most code |

I decided to implement the last method described in the table as it has no functional disadvantages whereas the other two have many and are far less efficient.

## Class definitions

### Node

#### Variables:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name/Type | Private/Protected/Public | What it does | When it’s changed | Example |
| size/Integer | Public | Contains the length of one side of the board of that node (one value as all boards are square) | Set at start, never changed after | 50 |
| Coordinates/List | Public | Stores the coordinates of that node for use in Dijkstra’s algorithm and graphical output | Set at start, never changed after | (100, 200) |
| population/Integer | Public | Stores the value of how many agents are currently in the node’s board | Whenever an agent leaves or enters the node | 2000 |
| board/Array | Public | Stores the board for the node, containing the positions of al the blanks and agents | Every tick | [[0, 0, 0, 1], [1, 0, “B”, 1], [“B”, “B”, 1, 0], [0, 1, 1, 0]] |
| unhappy/Array | Public | Contains the coordinates of all the unhappy tiles | Every tick | [(0, 0), (0, 1), (1, 1), (1, 2), (2, 3)] |
| blanks/Array | Public | Contains the coordinates of all the spaces agents can move to if they are unhappy (blanks) | Every tick | [(0, 0), (0, 1), (1, 1), (1, 2), (2, 3)] |
| squares/Integer | Public | Stores the total number of squares in the board | Set at start to size squared, never changed after | 2500 |
| bias/Integer | Public | Stores the percentage of neighbours each agent has to have at a minimum that are in the same group as them to be happy | Set at start by user, never changed after | 50 |
| diversity/Integer | Public | Stores the percentage of neighbours each agent has to have at a minimum that are in a different group to them to be happy | Set at start by user, never changed after | 20 |

#### Functions:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Arguments (+self) | What it does | Returns |
| genBoard | None | Generates the board | board, blanks |
| getUnhappy | None | Generates list of all the unhappy coordinates | unhappy |
| moveAgent | None | Moves a random unhappy space to a blank one and checks for any changes in happiness of affected spaces. Updates unhappy list | unhappy, board |

### Agent

#### Variables:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name/Type | Private/Protected/Public | What it does | When it’s changed | Example |
| currentNode/Class? | Public | Stores the instance of the node the agent is currently in | When the agent is unhappy for too long and moves to a different node. Changes to become each new node the agent passes on its way to the end node so new directions can be calculated | N/A |
| Coordinates/Array | Public | Stores the position of the agent in the current board it’s in | When the agent is moved | (15, 4) |
| Speed/Integer | Public | Stores the speed value of the agent, which is actually how slow it is so the higher the speed the slower it is | When slow motion or pause is selected | 5 |
| targetNode/Class? | Public | Stores the instance of the node the agent is moving to | When the agent moves or arrives to a destination | N/A |
| unhappyMoves/Integer | Public | Stores the number of unhappy moves the agent has made in a row | When the agent moves | 2 |
| threshold/Integer | Public | Stores the number of unhappy moves the agent is willing to put up with before moving to a new node. Only a variable for ease of change | Never | 3 |
| route/Array | Public | Takes the list of nodes/coordinates the agent needs to pass through to get to its new destination | When the node first starts to move and when it reaches the end | [[1,1], [2,2]…] |
| segregation/Float | Public | Stores the value of that agent’s ‘segregation percentage’, which is the number of agents of the same group around the agent/number of neighbours \* 100 | Whenever something around the agent changes or the agent moves | 50 |

#### Functions:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Arguments (+self) | What it does | Returns |
| getDir | currentNode, targetNode | Gets the distance between the coordinates of ‘currentNode’ and ‘targetNode’ and uses the agent’s speed to calculate the amount the agent has to move each tick | x and y values |
| checkMove | none | Checks if the agent’s ‘unhappyMoves’ value exceeds the agent’s ‘threshold’ value. Calls on ‘getMove’ if it is true | None |
| getMove | nodes | Selects a random node to move to and calls on ‘getDir’ to get the direction | The node to move to |
| updateMove | nodes? | Checks if the agent has reached its next checkpoint in its Dijkstra’s algorithm route. If it has, ‘getDir’ is called for new directions and ‘currentNode’ and ‘targetNode’ are changed | currentNode and targetNode |
| getSegregation | board? | Calculates ‘segregation’ for the agent | Segregation |

1. A map is a set of nodes connected by ‘edges’ of a certain length [↑](#footnote-ref-1)