CT421 Assignment 2 – Cristina Wall (20438522)

GitHub: https://github.com/Cristina-Wall/CT421_Assignment2

Part 1:

The graph topology that I have chosen is a small-world graph. I created this graph using a method called “generate_small_world_graph”, which called the watts_strogatz_graph() method in the networkx package. I used a graph with 100 nodes, 5 neighbours for each node, and a rewiring probability of 0.25. Initially I started with two colours and as the conflicts are calculated, the number of these increases until the program finds a solution with no conflicts.

The general structure of the program is as follows:

First creating a list of colours that will be used for the tree. There are two colours initially, so this is done by creating two random tuples, where each value is between 0 and 256. These represent RBG values, and each tuple will be used to create the colours.

Next the initial graph is generated. How this was done is covered in the introduction above.

A random colour was assigned to each node. The graph and the colour list are passed into a function called “assign_random_colors”. This adds a ‘color’ attribute to each node in the graph, which is randomly chosen from the colour list.

Plotting this initial graph gives us the graph in Fig. 1 below.

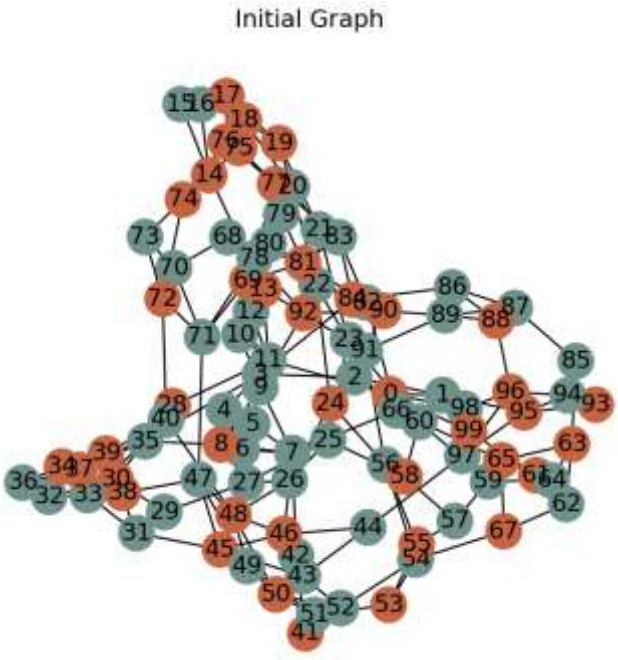

Figure 1: The initial graph.

We can see that since the colours are randomly assigned there are lots of conflicts between neighbouring nodes.

Next, the conflicts are counted by a “count_conflicts” method, which checks to see how many nodes have the same colour attribute as their neighbours.

In our initial graph in Fig. 1 above the number of conflicts is 190. As this is counting the conflicts for every node, this has counted each conflict twice, so dividing this by two gives us the number of edges that are connecting two nodes of the same colour. However, for the program we will stick with using the full doubled value.

After this, a copy of the initial graph is created and this new graph is passed to another function called “change_node_color”, which checks each node in the graph. For each node, the colours of all its neighbours are found and checked to see if there are any that have the same colour as itself. If there are, it checks to see if there are any colours that are not present in the neighbours of the node, and it changes its colour to this. If not and all colours are present in its neighbours, then it counts to see which colour is the least used in its neighbours and it switches to this node instead.

This process is repeated for 100 iterations, and if a graph with zero conflicts is not found after 100 iterations, then a new colour is added to the colour list and this process of changing the node colours starts again.

For the example used above in Fig. 1, the initial conflicts were 190 and after the first round of 100 iterations in which there are two colours available, the number of conflicts reduced to 116. After adding a third colour and repeating the 100 iterations again the number of conflicts reduced to 42, however the best solution found with three colours had only 38 conflicts. After adding a fourth colour, the number of iterations was reduced to 18, and the best solution found with four colours had only 6 conflicts. And finally at five colours, the program reached iteration number 14 and here it found a graph solution with zero conflicts.

Fig. 2 below shows the final graph that was found with no conflicts.

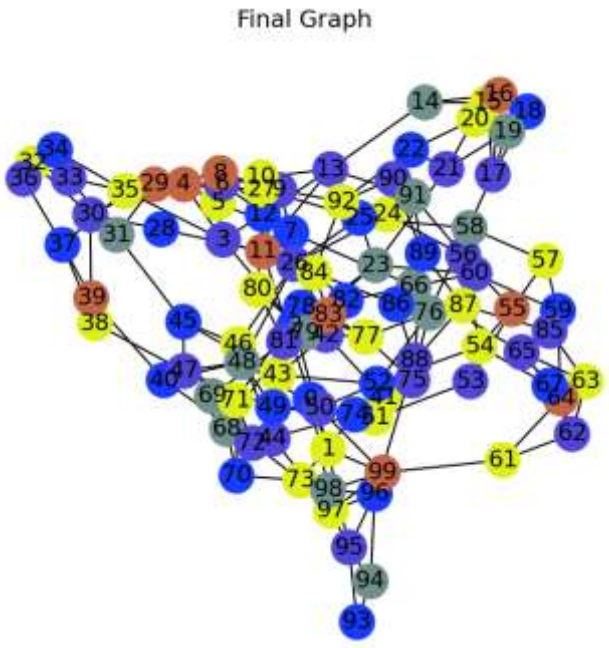

Figure 2: The final graph.

We can see that there are five different colours in this graph, blue, dark purple, brown, yellow, and grey.

Fig. 3 below shows the representation of the number of conflicts for each run.

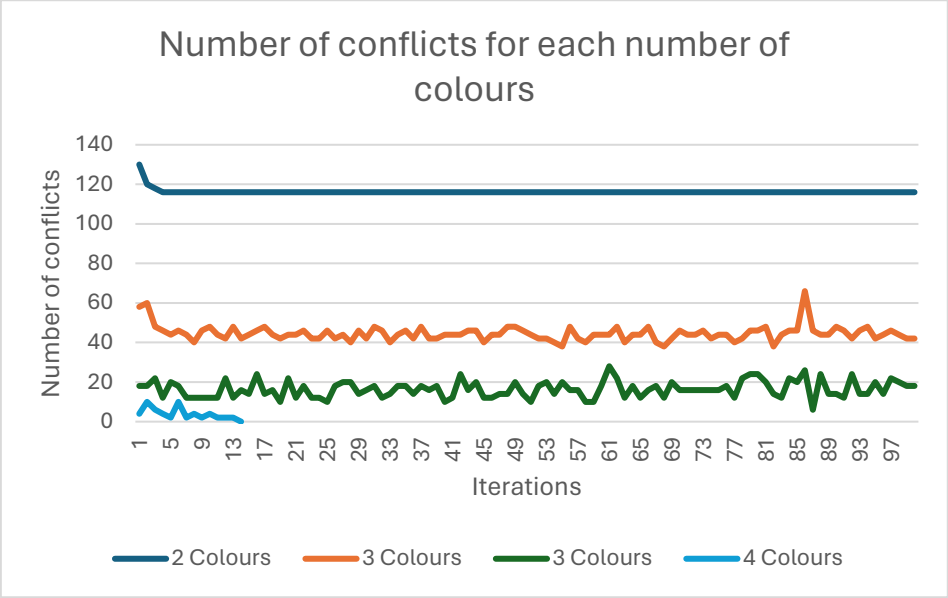

Figure 3: Graph of conflicts.

From this table we can see that around the first five iterations of each colour are the most significant in reducing the number of conflicts. After this, if a solution is not possible, the program keeps changing nodes to try and find a solution and ends up randomly increasing and decreasing the number of conflicts.

Part 2:

Research question: If a solution is found with no conflicts in a graph, how much of an impact will changing the colour of one random node have? How long will it take for the graph to be fixed after this situation?

This was done using a method called “change_random_node_color”, which picked one random node from the graph and changed it to a random colour in the list. It would then check if there are any conflicts and if there were none, a different random node would be chosen instead to change colour.

Once the node was changed and there were conflicts in the graph, this graph was passed back into the function used in part 1 that would go through each node and change the colour if it had a neighbour of the same colour.

The results of this are below.

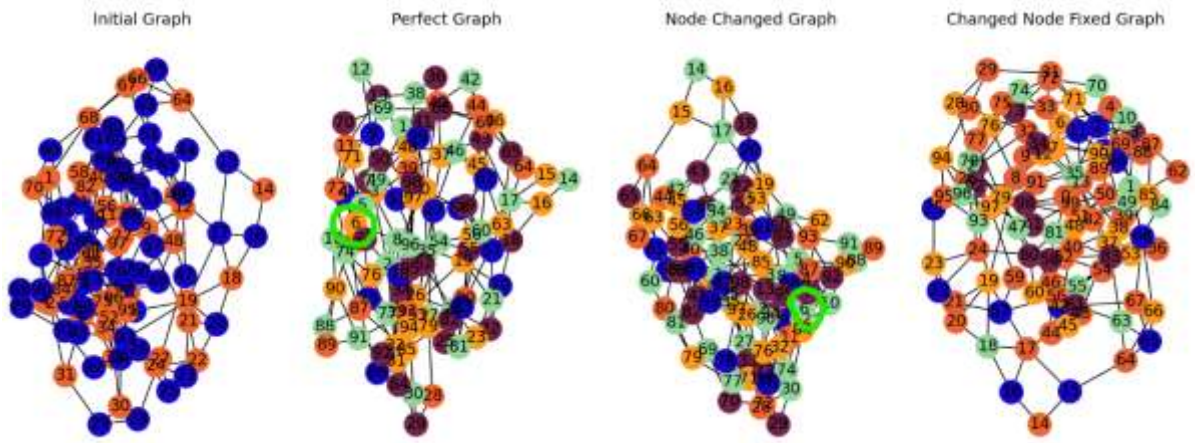

Figure 4: Graph sequence after changing a random node.

In Fig. 4, the “initial graph” is the one that the program originally comes up with, a small world graph with nodes assigned to only two different colours. The “perfect graph” is the solution found with no conflicts. The “node changed graph” is the graph after a random node has been changed. In this example node 39 was changed first. However, this did not produce any conflicts. Node 91 was then changed but this also did not produce any conflicts. Finally, node 6 was changed and this created six conflicts in the graph. Node 6 is highlighted in Fig.4 and the sequence of colour changes is shown in Fig. 5 below.

```
Node 39 was changed to color (0.9406803875818068, 0.3943801963474546, 0.16787065011446678)
Node 91 was changed to color (0.569265003276441, 0.8442980076311734, 0.6016189938034489)
Node 6 was changed to color (0.9902001216585711, 0.6053318559719555, 0.11283085656144953)
```

Figure 5: Output from changing random node colours.

The program only took one iteration to fix the graph and come up with another solution that has zero conflicts. This is the “changed node fixed graph” in Fig. 4.

Each time the program was run, it took only one iteration and never needed any additional colours to fix the graph. This was also tested for different sizes of graphs, and for graphs with different levels of connectedness, and for every combination it took only one iteration.

I also decided to test if 50% of the nodes were randomly changed, would it take longer to recover. 50 of the 100 nodes were changed and again, it produced the same result, taking only one iteration of the program to recover.

From these results, I concluded that once the graph has found a solution, it will take one iteration to fix any number of nodes being changed. As the number of colours required has already been found, it is able to quickly run through the graph and fix any conflicts.