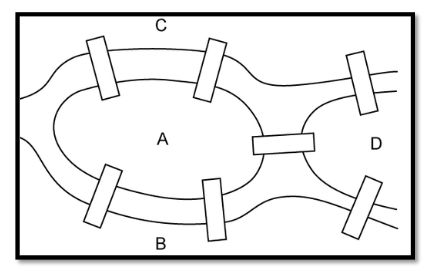
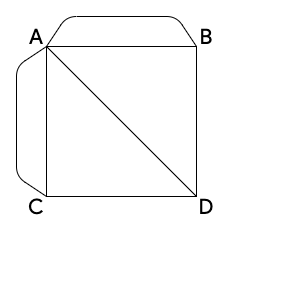
**Part A**

**Q-A1.1**

Given,



Now let’s convert the given image into a graph, with all the lands as vertices and bridges as edges



Note

* A graph has an Euler circuit if and only if the degree of every vertex is even.
* A graph has an Euler path if and only if there are at most two vertices with odd degree.

**Q-A1.1-a**

**Can the people of Konigsberg successfully walk over all the bridges once and get?**

**back to where they started?**

**Answer:** No, they can’t travel through all the bridges once and get back to where they started as the graph, **we considering is as a Euler Graph**

**Justification:**

The Degree of the Vertices i.e. **A = 5, B = 3, C = 3 and D = 3** but to find the **circuit to return back to same point** the Degree of the Vertices should be even not odd in order to find euler circuit

**Q-A1.1-b**

**Can the bridge walk be achieved if the people were happy not returning to their starting point?**

**Answer:** No, the bridge walk cannot be achieved if the people were happy not returning to their starting point as the graph, **we considering is as a Euler Graph**

**Justification:**

The Degree of the Vertices i.e., **should be odd and total number of odd vertices should be <= 2.** Clearly, all the vertices are odd so there are **4 odd vertices in the given graph no path is possible.**

**Q-A1.1-c**

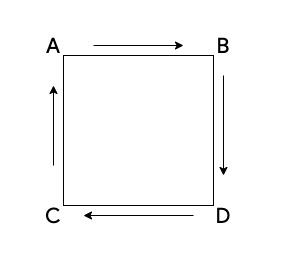
**If one or more of the bridges were removed, can the round trip walk around the bridges of Konigsberg be achieved?**

**Answer:** Yes, the round bridge walk can be achieved if we remove 3 edges from the existing graph and get back to where they started as the graph, **we considering is as a Euler Graph**

**Justification:**

Given Graph, The Degree of the Vertices i.e. **A = 5, B = 3, C = 3 and D = 3** but to find the **circuit to return back to same point** the Degree of the Vertices should be even not odd in order to find euler circuit, (Figure 2)

**Updated Graph**

****

As you can that we have removed the parallel edges between a vertices AB and AC and removed a diagonal edge from AD now the Graph as a Euler Circuit as, The Degree of the Vertices i.e. **A = 2, B = 2, C = 2 and D = 2**

**Q-A1.1-d**

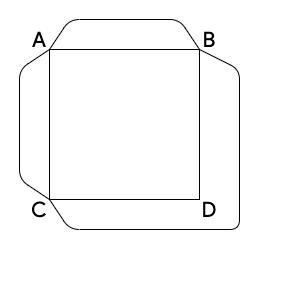
**If the city of Konigsberg had seven bridges arranged in some other way, will it be possible to make the round trip walk successfully?**

**Answer:** Yes, the round bridge walk can be achieved if we modify the edge that is traversing from AD from the existing graph to BC in the Updated Graph and get back to where they started as the graph, **we considering is as a Euler Graph**

**Justification:**

The Degree of the Vertices i.e. **A = 5, B = 3, C = 3 and D = 3** but to find the **circuit to return back to same point** the Degree of the Vertices should be even not odd in order to find euler circuit (Figure 2)

**Updated Graph**

****As you can see, we have **updated the graph and replaced the vertex from AD from the original graph to BC** in the new graph. Now, in order to **find the circuit**, we have to look at the **Degree of the Vertices** and they have to be even so, **A = 4, B = 4, C = 4 and D = 2**

**Now the updated graph has a Euler Circuit.**

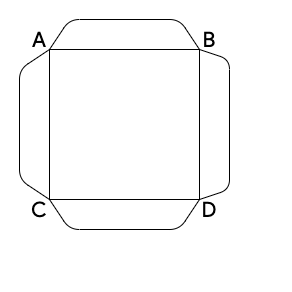
**Q-A1.1-e**

**If the seven bridges of Konigsberg are replaced with eight, nine and ten bridges, what can be commented about the land masses in each of the cases?**

**Answer:** Yes, the round bridge walk can be achieved if we modify the edge that is traversing from AD from the existing graph to BC in the Updated Graph and get back to where they started as the graph, **we considering is as a Euler Graph**

**Justification:**

**Case 1:** If the seven bridges of Konigsberg **are replaced with Eight, yes, the Euler Circuit and Euler Path are possible.**

****

**Condition:**

**Euler Circuit:**

All the vertices degree should be even, right? So, in the new graph **A = 4, B = 4, C = 4 and D = 4.** So, this has a **Euler Circuit**

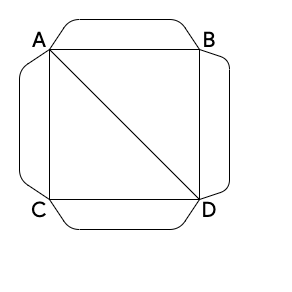
**Euler Path:**

The Sum of odd degree should be less than or equal 2**.. Here in there are 0.** So this has a **Euler Path**

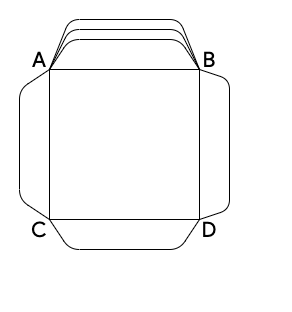
**Case 2:** If the seven bridges of Konigsberg **are replaced with Nine. The Euler Circuit is not possible but Euler Path is possible.**

**Euler Path:**

The Sum of odd degree should be less than or equal 2**. Here in there are 2.** So, this has a **Euler Path as Vertices A and D are odd vertices with degree 5.**

****

**Case 3:** If the seven bridges of Konigsberg **are replaced with Ten, yes, the Euler Circuit and Euler Path are possible.**

****

**Condition:**

**Euler Circuit:**

All the vertices degree should be even, right? So, in the new graph **A = 6, B = 6, C = 4 and D = 4.** So, this has a **Euler Circuit**

**Euler Path:**

The Sum of odd degree should be **less than or equal 2. Here in there are 0.** So, this has a **Euler Path**

**Q-A1.1-f**

**To cover n number of bridges is there a generalized result? Is there any relevance of knowing in what way the bridges are connected to the land masses?**

**Answer:** To generalized result of n number of bridges we can. So the condition is **If we have degrees of odd number of vertices and even number of vertices the we can generalise it,**

) + ) = )

Where **“d(vi)”** is **even number of edge at a vertex** and **“d(vj)”** is **odd number of edge at a vertex.** Now in order to find the number of bridges in a general form it would be,

) = 2 E

**where “E” is No of Bridges.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | B | C | D |
| A | 0 | 2 | 2 | 1 |
| B | 2 | 0 | 0 | 1 |
| C | 2 | 0 | 0 | 1 |
| D | 1 | 1 | 1 | 0 |

Given the Initial Graph we can find out how many vertices are there and how many edges by constructing an adjacency matrix,

**Q-A1.2**

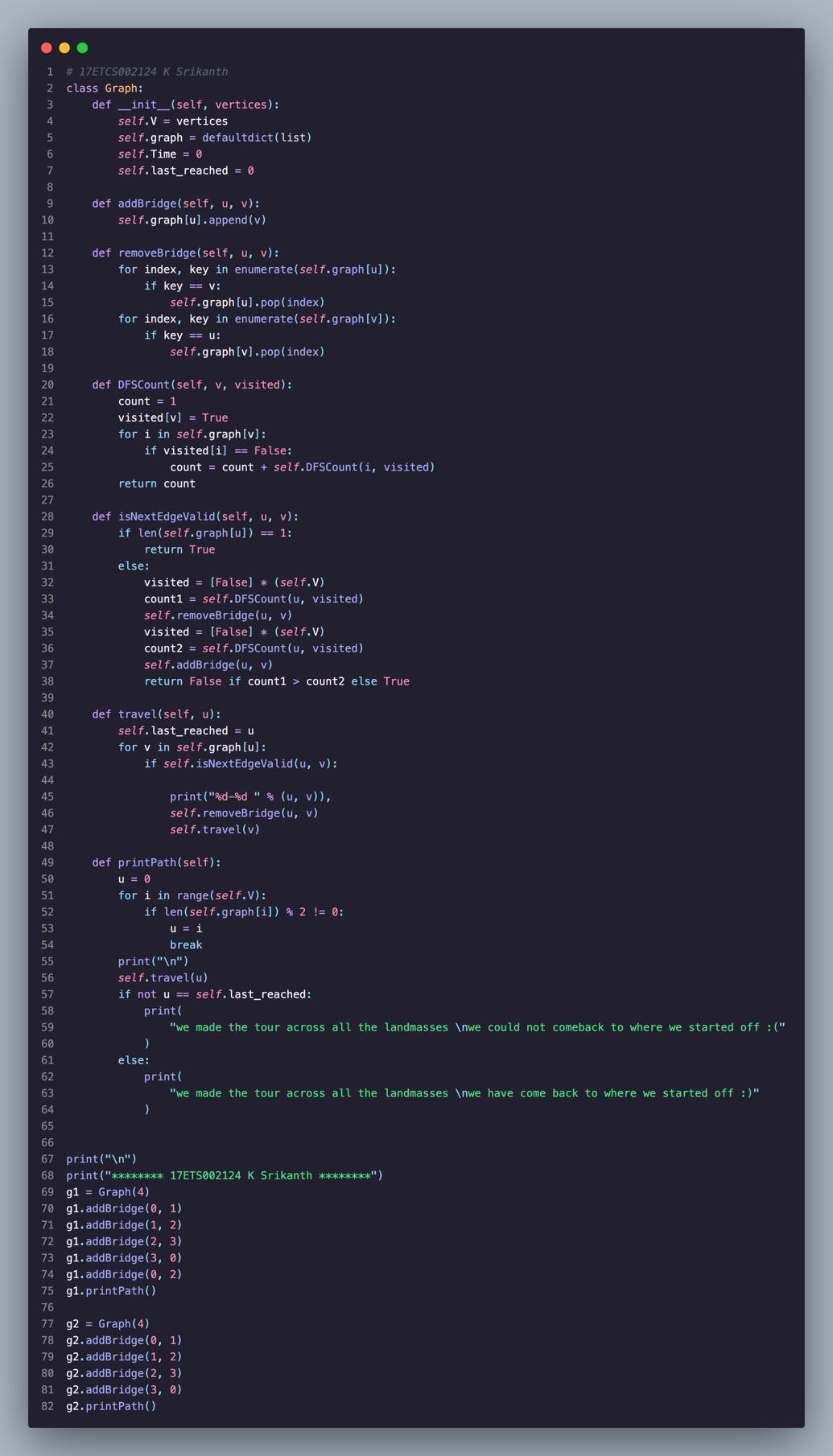
**Q-A1.2-a**

**Algorithm**

1. Start
2. Declare a Class (Graph)
   1. Declaring Instance of the class vertices (v)
   2. Declaring Instance of the class graph(\* which is a defaultdict)
   3. Declaring Instance of the class time
   4. Declaring Instance of the class last\_reached.
3. Declaring Function **(addBridge)** with params (u,v)
   1. Add(v) to graph (u)
4. Declaring Function **(removeBridge)** with params (u,v)
   1. For loop begins (index, key in enumerate of graph(element))
      1. If condition (key == v)
         1. Pop the element from the graph
   2. For loop begins (index, key in enumerate of graph(element))
      1. If condition (key == u)
         1. Pop the element from the graph
5. Declaring Function **(DFC Count)** with params (v, visited)
   1. Initialize count equals to 1
   2. Make visited of element true
   3. For loop beings (for element in graph visited)
      1. If visited element then make it false
         1. count += Instance of DFSCount(i, visited)
   4. Return count
6. Declaring Function (**isNextEdgeValid)** with params (u,v)
   1. If length of graph[u] is 1
      1. Return true
   2. Else
      1. Make Visisted false for all the instances of V
      2. Count1 += Instance of DFSCount(i, visited)
      3. Remove the bridge at (u,v)
      4. Make Visisted false for all the instances of V
      5. Count2 += Instance of DFSCount(i, visited)
      6. Add the bridge at (u,v)
      7. Return false if count 1 > count 2 else make it true
7. Declaring function (travel) with prams (u)
   1. Initialize instance of last\_reached as u
   2. For Loop Begins ( v in graph [u] )
      1. If is isNextEdgeValid (u,v)
         1. Display the path
         2. Remove the Bridge
         3. Continue traveling
8. Declaring function (printpath)
   1. Initializing u as 0
   2. For loop begins ( in in v)
      1. Check If length of (graph [element] % 2 !=0)
         1. If yes then make u = element then break
   3. Calling instance of travel (u)
   4. If not u == last reached
      1. Display (we didn’t make it back to the point)
   5. Else
      1. Display (We made it back to the point)
9. Declare the graph object with its instances to create a bridge with addBridge function
10. Stop

**Q-A1.2-b**

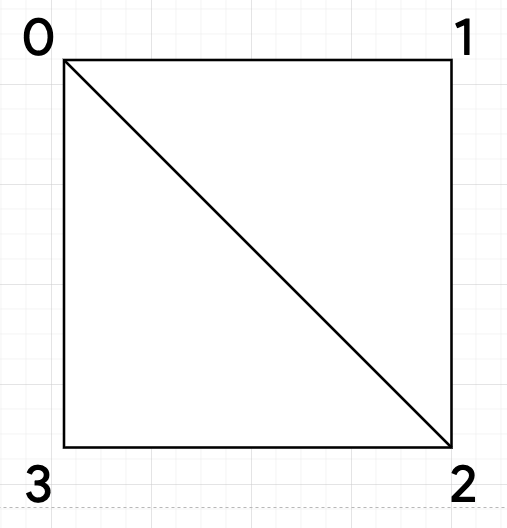
**Code**

****

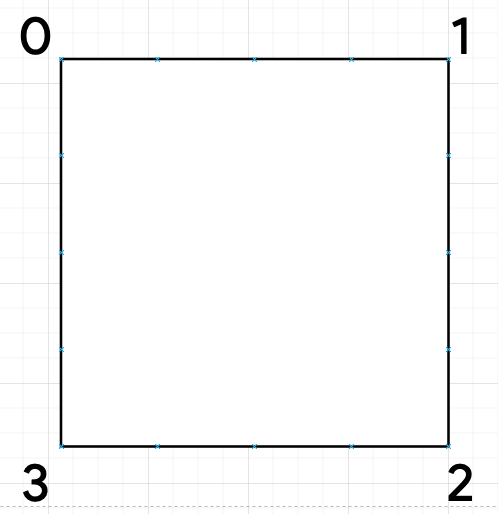
Python Code for the given problem statement

**Output**

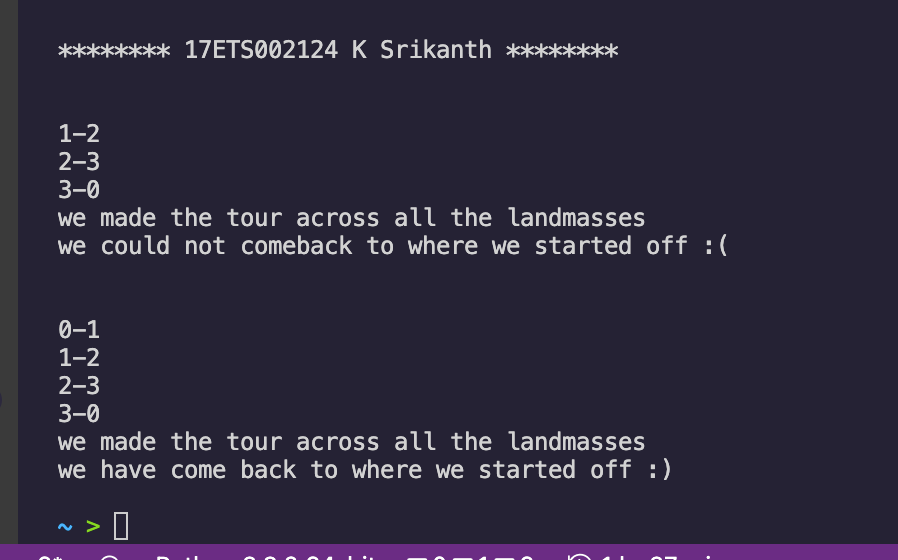
Here the Graph for G1 is, as an example is

****

Here the Graph for G2 is, as an example is

****

**Code Output**

****

Python Code output for the given problem statement

**Part B**

**Q-B1.1**

To Solve Sudoku using graph theory using coloring, we have to setup some ground rules in order to achieve that,

**Rule 1:** Let’s make sure that all the numbers in the sudoku puzzle i.e. {1,2,3,4,5,6,7,8,9} have independent color. So,

Let’s assume,

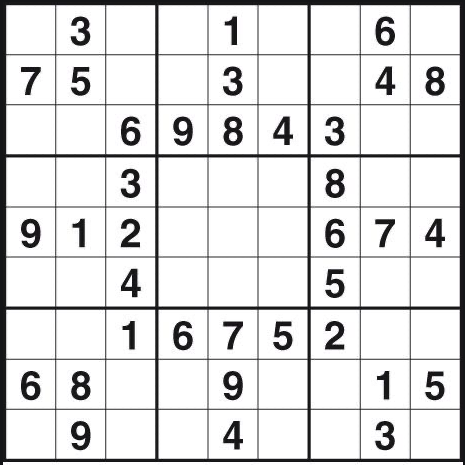
1. Number 1 is **Yellow**
2. Number 2 is **Green**
3. Number 3 is **Blue**
4. Number 4 is **Purple**
5. Number 5 is **Red**
6. Number 6 is **Black**
7. Number 7 is **Orange**
8. Number 8 is **Cyan**
9. Number 9 is **Magenta**

**Rule 2:** Now that we have organized our colors for our numbers. The Rule in order to solve our Sudoku is to make sure that that each number (i.e. color) from above should not be mapped to the same element in a row or column or the 3 X 3 Box that its currently in the state in

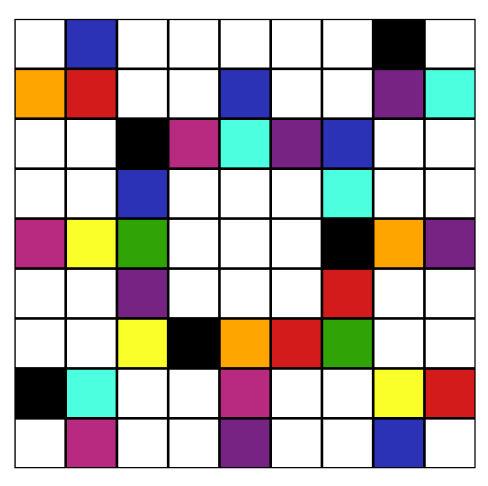
I.e.

Let’s take an example of a sudoku problem

Given,

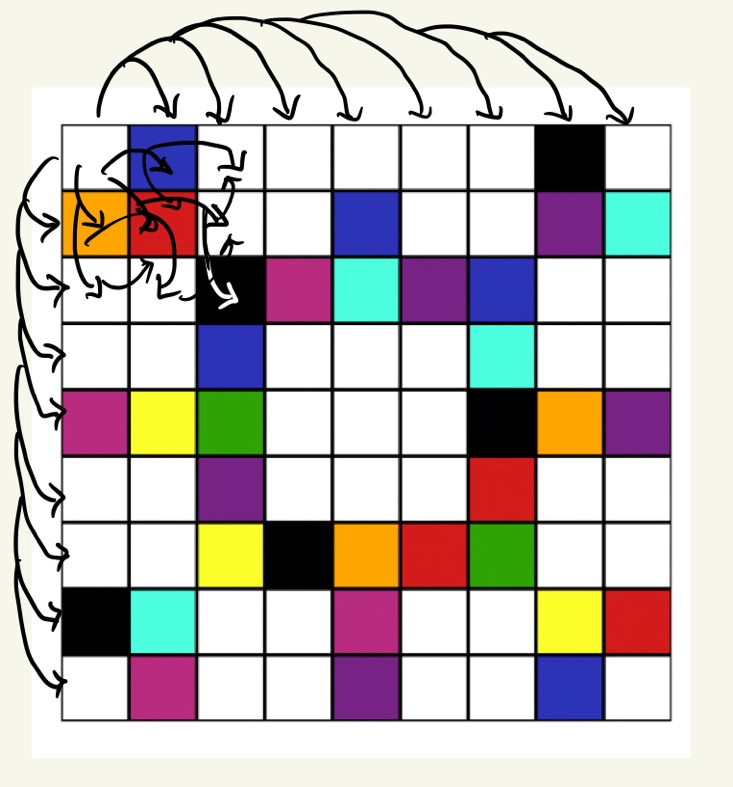
****

After Coloring it with Numbers,

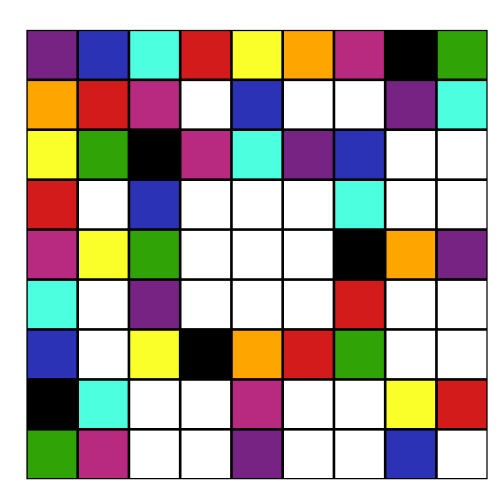
****

After we are done coloring out our number from the given problem statement the next thing, we do is map (make an edge) it. With its row and column and its 3 X 3 Box. We have to ensure that we don’t want to connect to the same color again as our connected nodes should be different from one and other.

Solving the First box,



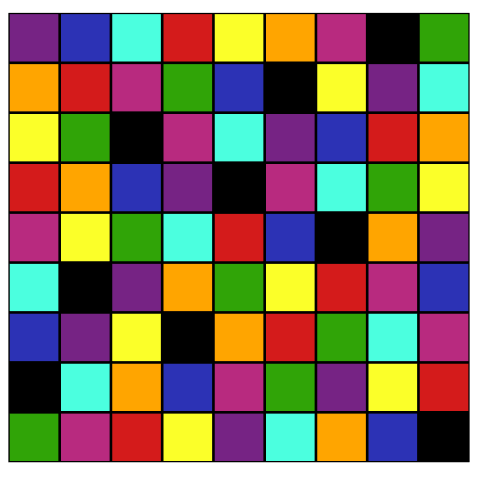
After connecting all the edges from one vertex our coloring graph will look something similar to this no, we just have to color the rest of the boxes in a way that don’t match up with the parent node that it is connected to with different colors. After coloring the First Box the Puzzle looks something like this,



Now that we have colored our first box where each color from the node is different when it is connected to other node.

And so, on we repeat our process over and over again to color everything out using this technique

**End Result**



Hence, we solved our sudoku problem with graph coloring technique

**Q-B1.2,**

**Algorithm**

1. **Start**
2. Create a recursive function that takescurrent vertex index, number of vertices and output colour array as arguments.
3. If the current vertex index is equal to number of vertices. Return True and print the colour configuration in output array.
4. Assign colour to a vertex (1 to 9).
5. For every assigned colour, check if the configuration is safe, (**i.e. check if the adjacent vertices do not have the same colour)** recursively call the function with next index and number of vertices
6. If any recursive function returns true break the loop and return true.
7. If no recursive function returns true, then return false.
8. **Stop**