Design overview:

As mentioned in the brief pseudo code in the previous document,We have built our complete pseudo code of implementation.

The coding is done in python with 2 sub-codes and 1 main code:

Connections.py(contains how to make connections as in grid as per constraints)
Graph.py(contains all nodes and their edges of representation)
Main.py(Contains graph coloring algorithm for solving sudoku)

We have to code and test the working of individual segments of all three parts.

Graph.py:

We create 2 classes: a node class and a graph class.

Node class:

Contains three portions id,data,connectedto
Initialize a node(self) using these 3 attributes
Id should be integer
Self.id = idx
Self.data = data
Self.connectedto = dict()
# Connected to is a variable dictionary which will store ids of nodes to which they are connected with
Inserting a neighbor to the node
neighbor= node object
Then we will add neighbour to the dictionary for checking with other elements
–> We define additional required functions such as getting elements data/info about the neighbor connected…

Graph class:

This is a normal graph algorithm
We initialize allnodes as in the dictionary,
Function for adding node to the graph
node= Node(idx=idx)
self.allNodes[idx] = node
Then we increase the number of vertices in the graph
Function for adding edge in graph:
source=edge where it starts from(node_id)
destiny=edge where it ends(node_id)
Then we will add that connection to the dictionary
self.allNodes[source].addNeighbor(self.allNodes[destiny],wt)
self.allNodes[destiny].addNeighbor(self.allNodes[source],wt)

This makes the graph undirected
These are the primitive functions for graph algorithms. We can add other functions as print nodes/DFS for searching..

Testing graph algorithm:
We need to check the working of the graph algorithm before coding the next portion.I have added BFS,DFS for checking whether the graph is traversing correctly/not.

Code:

```
def main() :
    g = Graph()
    for i in range(6) :
        g.addNode(i)
print("Vertices : ",g.getAllNodesIds())

    g.addEdge(src = 0, dst = 1, wt = 5)
    g.addEdge(0,5,2)
    g.addEdge(1,2,4)
    g.addEdge(2,3,9)
    g.addEdge(3,4,7)
    g.addEdge(3,5,3)
    g.addEdge(4,0,1)
    g.addEdge(5,4,8)
    g.addEdge(5,2,1)

    g.printEdges()

    print("DFS : (starting with 0)")
    g.DFS(0)
    print()

    print("BFS : (starting with 0)")
    g.BFS(0)
    print()
```

Results: Code has produced correct output

Connections.py:

The graph.py we have created is a normal graph algorithm but we have to deduce the algorithm to our requirement

We have to initialize the graph with grid conditions:
Self.graph = Graph()

```
  self.rows=9
  self.column=9
  self.blocks=81
for idx in range(1, self.total_blocks+1) :
          _ = self.graph.addNode(idx)
```

#we have 3 constraints which are:

1.Connecting Each Node to every other node in the same row

```
    for c in range(cols+1, 9) :
          row.append(matrix[rows][c])
     connections["rows"] = row
```

2.Connecting Each Node to every other node in the same column

```
     for r in range(rows+1, 9):
          col.append(matrix[r][cols])
     connections["cols"] = co
```

3.Connecting Each Node to every other node in the same sub grid(3*3 block)

Since we have already appended some of the edges in the 1,2. We will use if,else conditions for removing intersecting conditions

```
    if rows%3 == 0 :
          if cols%3 == 0 :
             block.append(matrix[rows+1][cols+1])
             block.append(matrix[rows+1][cols+2])
             block.append(matrix[rows+2][cols+1])
             block.append(matrix[rows+2][cols+2])
          elif cols%3 == 1 :
             block.append(matrix[rows+1][cols-1])
             block.append(matrix[rows+1][cols+1])
             block.append(matrix[rows+2][cols-1])
             block.append(matrix[rows+2][cols+1])
          elif cols%3 == 2 :
             block.append(matrix[rows+1][cols-2])
             block.append(matrix[rows+1][cols-1])
             block.append(matrix[rows+2][cols-2])
             block.append(matrix[rows+2][cols-1])
```

This for rows 3,6,9 we have to writes this for other rows using similar logic,

At last,

```
     connections["blocks"] = block
```

Testing connections.py:

For testing,we use the following code:

```
 def test_connections() :
    sudoku = SudokuConnections()
    sudoku.connectEdges()
    print("All node Ids : ")
    print(sudoku.graph.getAllNodesIds())
    print()
    for idx in sudoku.graph.getAllNodesIds() :
        print(idx, "Connected to->", sudoku.graph.allNodes[idx].getConnections())
test_connections()
```

Results: The code is giving correct results.

So till now, we have created the structure of Graph Class and then connected all the nodes.

Creating Main.py:

Here,we have to import the files created above

```
  from Connections import sudoku_Connections
```

We define board and give input in the code

```
board = [
          [0,0,0,4,0,0,0,0,0],
          [4,0,9,0,0,6,8,7,0],
          [0,0,0,9,0,0,1,0,0],
          [5,0,4,0,2,0,0,0,9],
          [0,7,0,8,0,4,0,6,0],
          [6,0,0,0,3,0,5,0,2],
          [0,0,1,0,0,7,0,0,0],
          [0,4,3,2,0,0,6,0,5],
          [0,0,0,0,0,5,0,0,0]
        ]
```

Then we write graph coloring algorithm for solving sudoku

We have to create a recursive function which takes input as graph then assigns colors to the graph.We will check whether the assigned configuration is true or not then we return true or false accordingly

```
color = [0] * (self.sudokuGraph.graph.totalV+1)
        given = [] # list of all the ids whos value is already given. Thus cannot be changed
        for row in range(len(self.board)) :
            for col in range(len(self.board[row])) :
                if self.board[row][col] != 0 :
                    #first get the idx of the position
                    idx = self.mappedGrid[row][col]
                    #update the color
                    color[idx] = self.board[row][col] # this is the main imp part
                    given.append(idx)
```

```
        return color, given

Function for checking configuration:
    if v in given and color[v] == c:
            return True
        elif v in given :
            return False
        for i in range(1, self.sudokuGraph.graph.totalV+1) :
            if color[i] == c and self.sudokuGraph.graph.isNeighbour(v, i) :
                return False
        return True
```

Testing:

For testing,I have used the code:

```
  s = SudokuBoard()
    print("BEFORE SOLVING ...")
    print("\n\n")
    s.printBoard()
    print("\nSolving ...")
    print("\n\n\nAFTER SOLVING ...")
    print("\n\n")
    s.solveGraphColoring(m=9)
    s.printBoard()
```

By this way,I have implemented backtracking for solving sudoku problem