

Implimentation:

This is the final code I have implemented for solving the problem of sudoku using backtracking
As we have done till so far there are three sub-segments of code

Graph.py:

```
class Node :
```

```
    def __init__(self, idx, data = 0) : # Constructor
        self.id = idx
        self.data = data
        self.connectedTo = dict()

    def addNeighbour(self, neighbour , weight = 0) :
        if neighbour.id not in self.connectedTo.keys() :
            self.connectedTo[neighbour.id] = weight

    # setter
    def setData(self, data) :
        self.data = data

    #getter
    def getConnections(self) :
        return self.connectedTo.keys()

    def getID(self) :
        return self.id

    def getData(self) :
        return self.data

    def getWeight(self, neighbour) :
        return self.connectedTo[neighbour.id]

    def __str__(self) :
        return str(self.data) + " Connected to : "+ \
            str([x.data for x in self.connectedTo])
```

```
class Graph :
```

```
    totalV = 0 # total vertices in the graph
    def __init__(self) :
```

```

        self.allNodes = dict()
def addNode(self, idx) :
    if idx in self.allNodes :
        return None
    Graph.totalV += 1
    node = Node(idx=idx)
    self.allNodes[idx] = node
    return node

def addNodeData(self, idx, data) :
    if idx in self.allNodes :
        node = self.allNodes[idx]
        node.setData(data)
    else :
        print("No ID to add the data.")

def addEdge(self, src, dst, wt = 0) :
    self.allNodes[src].addNeighbour(self.allNodes[dst], wt)
    self.allNodes[dst].addNeighbour(self.allNodes[src], wt)

def isNeighbour(self, u, v) :
    if u >=1 and u <= 81 and v >=1 and v <= 81 and u !=v :
        if v in self.allNodes[u].getConnections() :
            return True
    return False

def printEdges(self) :
    for idx in self.allNodes :
        node = self.allNodes[idx]
        for con in node.getConnections() :
            print(node.getID(), "--> ",
                  self.allNodes[con].getID())

# getter
def getNode(self, idx) :
    if idx in self.allNodes :
        return self.allNodes[idx]
    return None

def getAllNodesIds(self) :
    return self.allNodes.keys()

```

```

# methods
def DFS(self, start) :
    # STACK
    visited = [False]*Graph.totalV

    if start in self.allNodes.keys() :
        self.__DFSUtility(node_id = start, visited=visited)
    else :
        print("Start Node not found")

def __DFSUtility(self, node_id, visited) :
    visited = self.__setVisitedTrue(visited=visited, node_id=node_id)
    #print
    print(self.allNodes[node_id].getID(), end = " ")

    #Recursive Stack
    for i in self.allNodes[node_id].getConnections() :
        if visited[self.allNodes[i].getID()] == False :
            self.__DFSUtility(node_id = self.allNodes[i].getID(),
                               visited=visited)

def BFS(self, start) :
    #Queue
    visited = [False]*Graph.totalV

    if start in self.allNodes.keys() :
        self.__BFSUtility(node_id = start, visited=visited)
    else :
        print("Start Node not found")

def __BFSUtility(self, node_id, visited) :
    queue = []
    visited = self.__setVisitedTrue(visited=visited, node_id=node_id)

    queue.append(node_id)

    while queue != [] :
        x = queue.pop(0)
        #print
        print(self.allNodes[x].getID(), end = " ")

        for i in self.allNodes[x].getConnections() :
            idx = self.allNodes[i].getID()
            if visited[idx] == False :

```

```
queue.append(idx)
visited = self.__setVisitedTrue(visited=visited,
node_id=idx)
```

```
def __setVisitedTrue(self, visited, node_id) :
    visited[node_id] = True
    return visited
```

There are some additional functions in the code which are made to check whether graph is created correctly or not. Hence, functions like BFS, DFS and printing is not necessary in the code but added for testing. The segments as number of vertices can be given as global input in the graph.py itself but it is avoided for inclusion of all inputs and connections of grid in connections.py..

This code is been tested with some input with code:

```
def test() :
    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()
```

```
if __name__ == "__main__":
    test()
```

Output:

```
main.py
1 class Node :
2
3     def __init__(self, idx, data = 0) : # Constructor
4         self.id = idx
5         self.data = data
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
103
```

Hence the graph part of the code is working correctly..

The next part of the code is **sudoku_connections**

```
from graph import Graph
```

```
class SudokuConnections :
    def __init__(self) : # constructor
        self.graph = Graph() # Graph Object
        self.rows = 9
        self.cols = 9
        self.total_blocks = self.rows*self.cols #81
        self.__generateGraph() # Generates all the nodes
        self.connectEdges() # connects all the nodes acc to sudoku constraints
        self.allIds = self.graph.getAllNodesIds() # storing all the ids in a list
    def __generateGraph(self) :
        for idx in range(1, self.total_blocks+1) :
            _ = self.graph.addNode(idx)

    def connectEdges(self) :
        matrix = self.__getGridMatrix()

        head_connections = dict() # head : connections
```

```

for row in range(9) :
    for col in range(9) :
        head = matrix[row][col] #id of the node
        connections = self.__whatToConnect(matrix, row, col)
        head_connections[head] = connections
# connect all the edges

self.__connectThose(head_connections=head_connections)

def __connectThose(self, head_connections) :
    for head in head_connections.keys() : #head is the start idx
        connections = head_connections[head]
        for key in connections : #get list of all the connections
            for v in connections[key] :
                self.graph.addEdge(src=head, dst=v)

def __whatToConnect(self, matrix, rows, cols) :
    connections = dict()

    row = []
    col = []
    block = []

    # ROWS
    for c in range(cols+1, 9) :
        row.append(matrix[rows][c])

    connections["rows"] = row

    # COLS
    for r in range(rows+1, 9):
        col.append(matrix[r][cols])

    connections["cols"] = col

    # BLOCKS

    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])
```

```
elif rows%3 == 1 :
```

```
if cols%3 == 0 :
```

```
block.append(matrix[rows-1][cols+1])
block.append(matrix[rows-1][cols+2])
block.append(matrix[rows+1][cols+1])
block.append(matrix[rows+1][cols+2])
```

```
elif cols%3 == 1 :
```

```
block.append(matrix[rows-1][cols-1])
block.append(matrix[rows-1][cols+1])
block.append(matrix[rows+1][cols-1])
block.append(matrix[rows+1][cols+1])
```

```
elif cols%3 == 2 :
```

```
block.append(matrix[rows-1][cols-2])
block.append(matrix[rows-1][cols-1])
block.append(matrix[rows+1][cols-2])
block.append(matrix[rows+1][cols-1])
```

```
elif rows%3 == 2 :
```

```
if cols%3 == 0 :
```

```

        block.append(matrix[rows-2][cols+1])
        block.append(matrix[rows-2][cols+2])
        block.append(matrix[rows-1][cols+1])
        block.append(matrix[rows-1][cols+2])

    elif cols%3 == 1 :

        block.append(matrix[rows-2][cols-1])
        block.append(matrix[rows-2][cols+1])
        block.append(matrix[rows-1][cols-1])
        block.append(matrix[rows-1][cols+1])

    elif cols%3 == 2 :

        block.append(matrix[rows-2][cols-2])
        block.append(matrix[rows-2][cols-1])
        block.append(matrix[rows-1][cols-2])
        block.append(matrix[rows-1][cols-1])

    connections["blocks"] = block
    return connections

def __getGridMatrix(self) :
    matrix = [[0 for cols in range(self.cols)]
               for rows in range(self.rows)]

    count = 1
    for rows in range(9) :
        for cols in range(9):
            matrix[rows][cols] = count
            count+=1
    return matrix

```

In the part of the code, for avoiding the common cases which are included in row and column constraints are been avoided using if loop But using a for loop would have reduced the length of code but it will increase time complexity of calculating all cases hence I avoided that approach

I have tested this part of the code whether it is giving correct output or not using test code:

```

def test_connections() :
    sudoku = SudokuConnections()
    sudoku.connectEdges()
    print("All node lds : ")
    print(sudoku.graph.getAllNodesIds())

```



```

print()
for idx in sudoku.graph.getAllNodesIds() :
    print(idx, "Connected to->", sudoku.graph.allNodes[idx].getConnections())

if __name__ == "__main__" :
    test_connections()

```

Output:

It is providing correct output,

All node Ids :

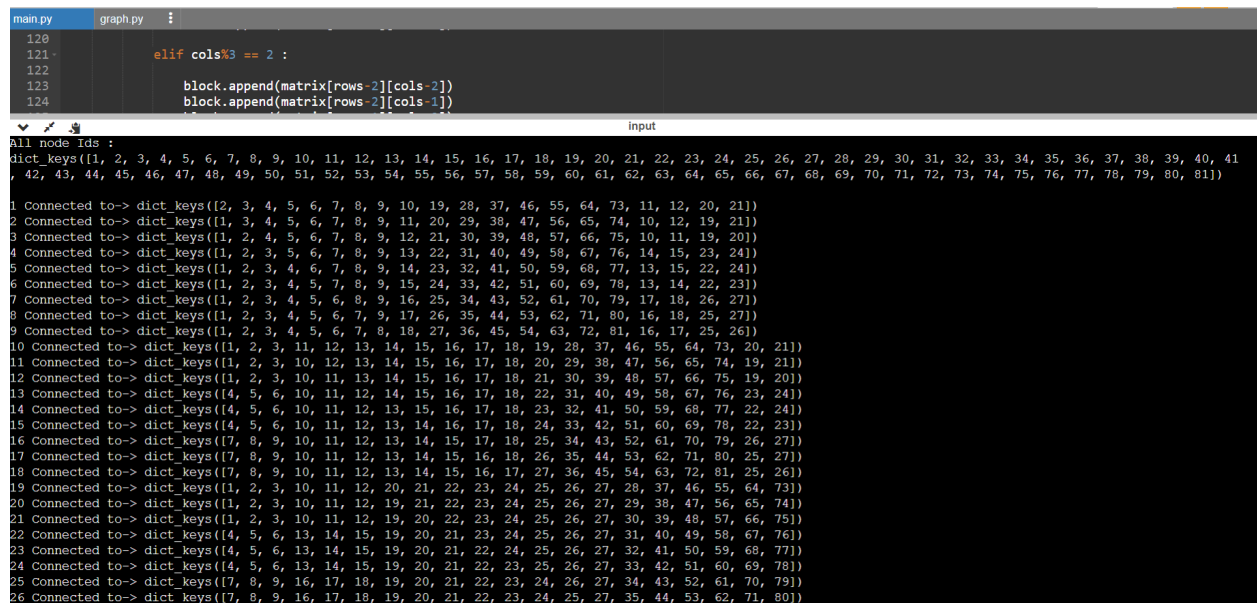
```
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81])
```

```

1 Connected to-> dict_keys([2, 3, 4, 5, 6, 7, 8, 9, 10, 19, 28, 37, 46, 55, 64, 73, 11, 12, 20, 21])
2 Connected to-> dict_keys([1, 3, 4, 5, 6, 7, 8, 9, 11, 20, 29, 38, 47, 56, 65, 74, 10, 12, 19, 21])
3 Connected to-> dict_keys([1, 2, 4, 5, 6, 7, 8, 9, 12, 21, 30, 39, 48, 57, 66, 75, 10, 11, 19, 20])

```

.....,



```

main.py graph.py
120
121 elif cols%3 == 2 :
122
123     block.append(matrix[rows-2][cols-2])
124     block.append(matrix[rows-2][cols-1])

```

input

```

All node Ids :
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81])

1 Connected to-> dict_keys([2, 3, 4, 5, 6, 7, 8, 9, 10, 19, 28, 37, 46, 55, 64, 73, 11, 12, 20, 21])
2 Connected to-> dict_keys([1, 3, 4, 5, 6, 7, 8, 9, 11, 20, 29, 38, 47, 56, 65, 74, 10, 12, 19, 21])
3 Connected to-> dict_keys([1, 2, 4, 5, 6, 7, 8, 9, 12, 21, 30, 39, 48, 57, 66, 75, 10, 11, 19, 20])
4 Connected to-> dict_keys([1, 2, 3, 5, 6, 7, 8, 9, 13, 22, 31, 40, 49, 58, 67, 76, 14, 15, 23, 24])
5 Connected to-> dict_keys([1, 2, 3, 4, 6, 7, 8, 9, 14, 23, 32, 41, 50, 59, 68, 77, 13, 15, 22, 24])
6 Connected to-> dict_keys([1, 2, 3, 4, 5, 7, 8, 9, 15, 24, 33, 42, 51, 60, 69, 78, 13, 14, 22, 23])
7 Connected to-> dict_keys([1, 2, 3, 4, 5, 6, 8, 9, 16, 25, 34, 43, 52, 61, 70, 79, 17, 18, 26, 27])
8 Connected to-> dict_keys([1, 2, 3, 4, 5, 6, 7, 9, 17, 26, 35, 44, 53, 62, 71, 80, 16, 18, 25, 27])
9 Connected to-> dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 18, 27, 36, 45, 54, 63, 72, 81, 16, 17, 25, 26])
10 Connected to-> dict_keys([1, 2, 3, 11, 12, 13, 14, 15, 16, 17, 18, 19, 28, 37, 46, 55, 64, 73, 20, 21])
11 Connected to-> dict_keys([1, 2, 3, 10, 12, 13, 14, 15, 16, 17, 18, 20, 29, 38, 47, 56, 65, 74, 19, 21])
12 Connected to-> dict_keys([1, 2, 3, 10, 11, 13, 14, 15, 16, 17, 18, 21, 30, 39, 48, 57, 66, 75, 19, 20])
13 Connected to-> dict_keys([4, 5, 6, 10, 11, 12, 14, 15, 16, 17, 18, 22, 31, 40, 49, 58, 67, 76, 23, 24])
14 Connected to-> dict_keys([4, 5, 6, 10, 11, 12, 13, 15, 16, 17, 18, 23, 32, 41, 50, 59, 68, 77, 22, 24])
15 Connected to-> dict_keys([4, 5, 6, 10, 11, 12, 13, 14, 16, 17, 18, 24, 33, 42, 51, 60, 69, 78, 22, 23])
16 Connected to-> dict_keys([7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 25, 34, 43, 52, 61, 70, 79, 26, 27])
17 Connected to-> dict_keys([7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 26, 35, 44, 53, 62, 71, 80, 25, 27])
18 Connected to-> dict_keys([7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 27, 36, 45, 54, 63, 72, 81, 25, 26])
19 Connected to-> dict_keys([1, 2, 3, 10, 11, 12, 20, 21, 22, 23, 24, 25, 26, 27, 28, 37, 46, 55, 64, 73])
20 Connected to-> dict_keys([1, 2, 3, 10, 11, 12, 19, 21, 22, 23, 24, 25, 26, 27, 29, 38, 47, 56, 65, 74])
21 Connected to-> dict_keys([1, 2, 3, 10, 11, 12, 19, 20, 22, 23, 24, 25, 26, 27, 30, 39, 48, 57, 66, 75])
22 Connected to-> dict_keys([4, 5, 6, 13, 14, 15, 19, 20, 21, 23, 24, 25, 26, 27, 31, 40, 49, 58, 67, 76])
23 Connected to-> dict_keys([4, 5, 6, 13, 14, 15, 19, 20, 21, 22, 24, 25, 26, 27, 32, 41, 50, 59, 68, 77])
24 Connected to-> dict_keys([4, 5, 6, 13, 14, 15, 19, 20, 21, 22, 23, 25, 26, 27, 33, 42, 51, 60, 69, 78])
25 Connected to-> dict_keys([7, 8, 9, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 34, 43, 52, 61, 70, 79])
26 Connected to-> dict_keys([7, 8, 9, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 35, 44, 53, 62, 71, 80])

```

Finally all these parts of code are merged under main function

```
from sudoku_connections import SudokuConnections
```

```

class SudokuBoard :
    def __init__(self) :

```

```

self.board = self.getBoard()

self.sudokuGraph = SudokuConnections()
self.mappedGrid = self.__getMappedMatrix() # Maps all the ids to the position in the matrix

def __getMappedMatrix(self) :
    matrix = [[0 for cols in range(9)]
               for rows in range(9)]

    count = 1
    for rows in range(9) :
        for cols in range(9):
            matrix[rows][cols] = count
            count+=1
    return matrix

def getBoard(self) :
    return board

def printBoard(self) :

    print("  1 2 3   4 5 6   7 8 9")
    for i in range(len(self.board)) :
        if i%3 == 0 :#and i != 0:
            print(" ----- ")

        for j in range(len(self.board[i])) :
            if j %3 == 0 :#and j != 0 :
                print("| ", end = "")
            if j == 8 :
                print(self.board[i][j], " | ", i+1)
            else :
                print(f"{ self.board[i][j] } ", end="")
        print(" ----- ")

def is_Blank(self) :

    for row in range(len(self.board)) :
        for col in range(len(self.board[row])) :
            if self.board[row][col] == 0 :
                return (row, col)
    return None

```

```

def graphColoringInitializeColor(self):
    """
    fill the already given colors
    """
    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

def solveGraphColoring(self, m =9) :

    color, given = self.graphColoringInitializeColor()
    if self.__graphColorUtility(m =m, color=color, v =1, given=given) is None :
        print(":(")
        return False
    count = 1
    for row in range(9) :
        for col in range(9) :
            self.board[row][col] = color[count]
            count += 1
    return color

def __graphColorUtility(self, m, color, v, given) :

    if v == self.sudokuGraph.graph.totalV+1 :
        return True
    for c in range(1, m+1) :
        if self.__isSafe2Color(v, color, c, given) == True :
            color[v] = c
            if self.__graphColorUtility(m, color, v+1, given) :
                return True
    if v not in given :
        color[v] = 0

def __isSafe2Color(self, v, color, c, given) :

    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

```

I have given input of the code as:

```

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]
]

```

It has provided correct output,I have verified it using:

```

def main() :
    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()

if __name__ == "__main__" :
    main()

```

Output:

It provided output as:

```

1 2 3   4 5 6   7 8 9
-----
| 1 8 5 | 4 7 3 | 9 2 6 | 1
| 4 2 9 | 5 1 6 | 8 7 3 | 2
| 3 6 7 | 9 8 2 | 1 5 4 | 3
-----

```

534	621	789	4
972	854	361	5
618	739	542	6

251	367	498	7
743	298	615	8
896	145	237	9

This output is correct...

```

input
BEFORE SOLVING ...

  1 2 3      4 5 6      7 8 9
- - - - -
| 0 0 0 | 4 0 0 | 0 0 0 | 1
| 4 0 9 | 0 0 6 | 8 7 0 | 2
| 0 0 0 | 9 0 0 | 1 0 0 | 3
- - - - -
| 5 0 4 | 0 2 0 | 0 0 9 | 4
| 0 7 0 | 8 0 4 | 0 6 0 | 5
| 6 0 0 | 0 3 0 | 5 0 2 | 6
- - - - -
| 0 0 1 | 0 0 7 | 0 0 0 | 7
| 0 4 3 | 2 0 0 | 6 0 5 | 8
| 0 0 0 | 0 0 5 | 0 0 0 | 9
- - - - -

Solving ...

AFTER SOLVING ...

  1 2 3      4 5 6      7 8 9
- - - - -
| 1 8 5 | 4 7 3 | 9 2 6 | 1
| 4 2 9 | 5 1 6 | 8 7 3 | 2
| 3 6 7 | 9 8 2 | 1 5 4 | 3
- - - - -
| 5 3 4 | 6 2 1 | 7 8 9 | 4
| 9 7 2 | 8 5 4 | 3 6 1 | 5
| 6 1 8 | 7 3 9 | 5 4 2 | 6
- - - - -
| 2 5 1 | 3 6 7 | 4 9 8 | 7

```

By this way We have solved the problem of sudoku using graph coloring concept...