DELHI TECHNOLOGICAL UNIVERSITY

GRAPH THEORY (MC-405)

Midterm Innovative Project

SUBMITTED BY:

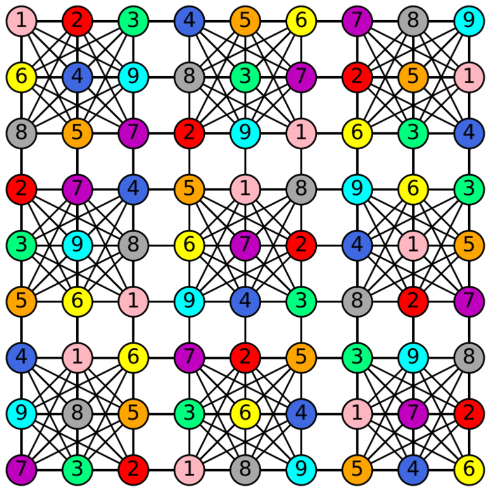
Aiman Siddiqua - 2K18/MC/008

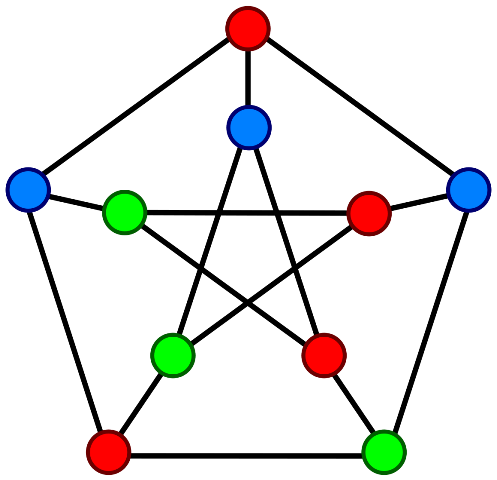
Apoorva - 2K18/MC/019

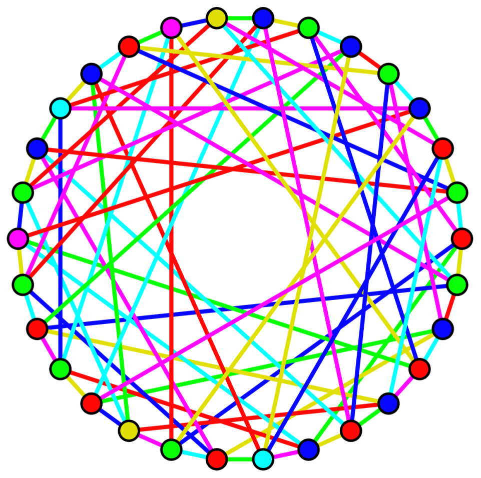
SUBMITTED TO:

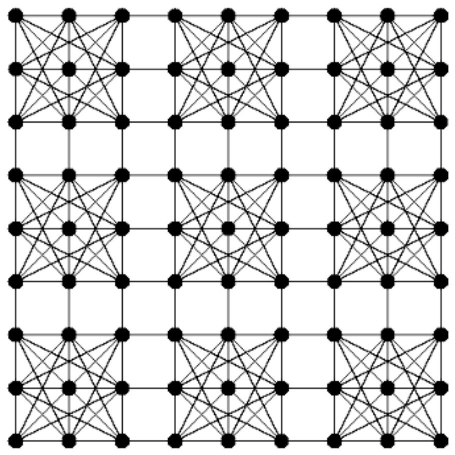
Dr. Sangita Kansal

SOLVING SUDOKU: An Application of Graph Coloring





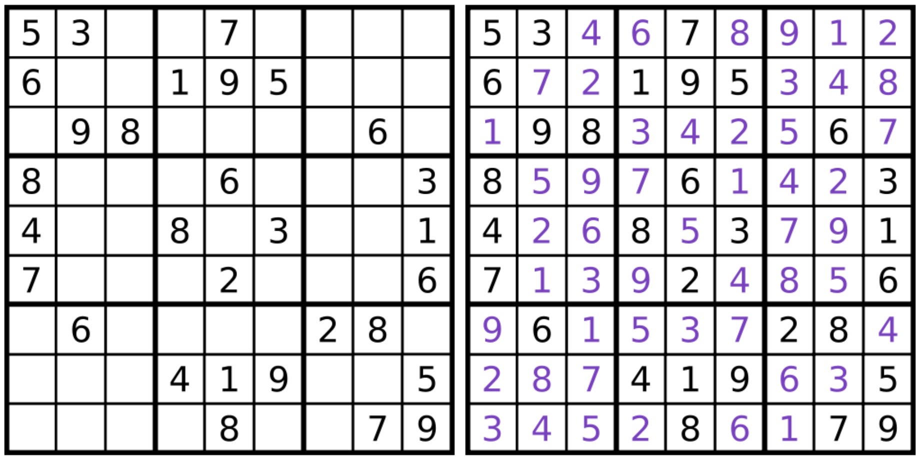




INDEX

INTRODUCTION

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic sudoku, the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid contain all of the digits from 1 to 9.



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

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

While it is relatively easy to solve a sudoku puzzle by a brute-force algorithm, we can find a more elegant solution by modelling it as a graph colouring problem.

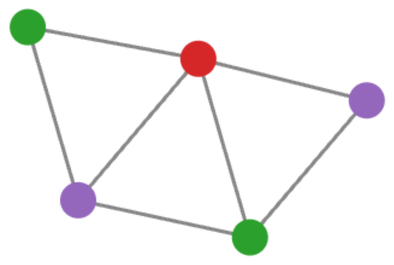
We assume the grid squares are vertices, and the numbers are colours that must be different if in the same row, column, or 3×3 grid (an edge connects such vertices in the graph). The sudoku is then a graph of 81 vertices and chromatic number 9.

The inspiration to solve a sudoku puzzle comes from a study in Graph Theory. In 2007, the mathematicians Herzberg and Murty developed a method to calculate the number of ways one can colour the vertices of a partially coloured graph so that no two adjacent vertices have the same colour. This result was not developed with puzzles in mind, but it applies immediately to Sudoku puzzles once one has recognized that a Sudoku puzzle can be presented as a graph coloring problem.

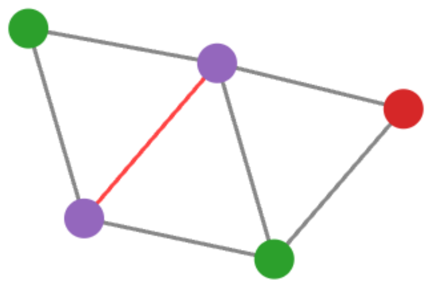
In this project, we shall attempt to learn about graph coloring and apply the same to solve Sudoku puzzles.

GRAPH COLOURING

Graph Colouring is the assignment of colors to each vertex in a graph such that no two adjacent vertices get the same color. Sometimes, this is also referred to as **proper colouring** of the graph.



This graph is properly colored

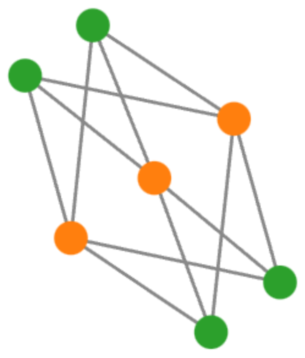


This graph is not properly colored

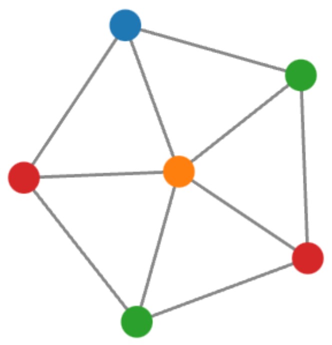
In modern times, many open problems in algebraic graph theory deal with the relation between chromatic polynomials and their graphs. Applications for solved problems have been found in areas such as computer science, information theory, and complexity theory. Many day-to-day problems, like minimizing conflicts in scheduling, are also equivalent to graph colorings.

K-Colourable Graph

A graph is said to be k-colorable if it can be properly colored using k colors.
For example, a **bipartite graph** is 2-colorable.



This graph is 2-colorable



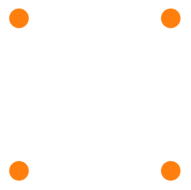
This graph is 4-colorable

Chromatic Number

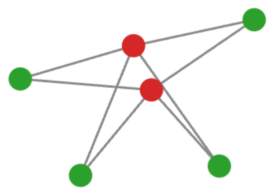
Chromatic Number of a graph is the minimum value of k for which the graph is k-colorable. It is the minimum number of colors needed for a proper coloring of the graph. Chromatic Number of a graph G is denoted by $\chi(G)$.

Some common types of graphs and their chromatic number:

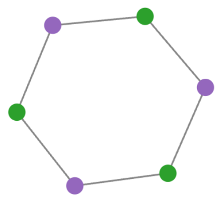
Empty Graph
It is a graph without any edges. All the vertices are isolated.
$\chi(G) = 1$.



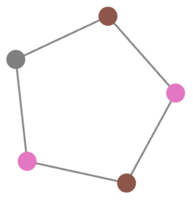
Bipartite Graph
A non-empty bipartite graph has $\chi(G) = 2$.



Cycle Graph
A cycle graph of order n is denoted by C_n. A cycle graph of odd order has $\chi(C_{2n+1}) = 3$ and that of even order has $\chi(C_{2n}) = 2$. Hence a cycle of even order is a bipartite graph.

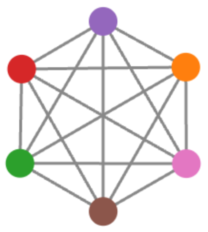


Cycle graph of even order



Cycle graph of odd order

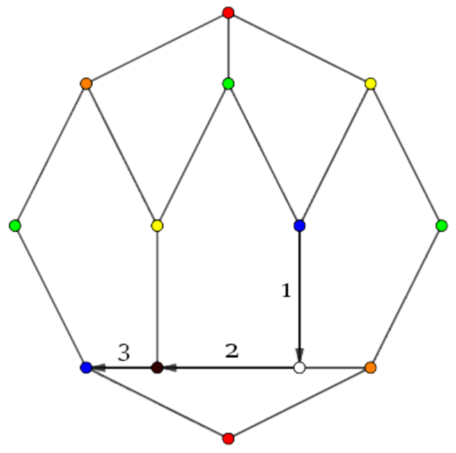
Complete Graph
Since each vertex is connected to every other vertex in a complete graph, $\chi(Kn) = n$.



K-Distant Graph Coloring

A 2-distant graph colouring of some graph is a way that we can colour in each vertex in the graph such that no two neighbouring vertices have the same colour. That is, any two vertices with the same colour must be at least 2 vertices away from each other.

Now, a k-distant graph colouring, for any positive integer k, is a way that we can colour in each vertex in the graph such that any two vertices which have the same colour are at least k vertices away from each other.

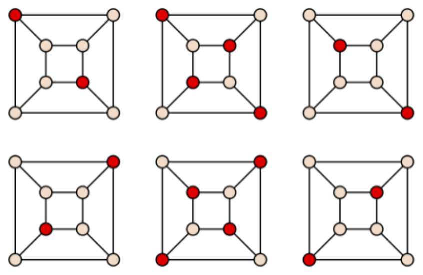


3-distant graph coloring

Maximal Independent Set

In graph theory, an **independent set** is a set of vertices in a graph, no two of which are adjacent. A **maximal independent set** (MIS) or **maximal stable set** is an independent set that is not a subset of any other independent set. In other words, there is no vertex outside the independent set that may join it because it is maximal with respect to the independent set property.

The graph of a cube has six different independent sets which have been depicted by red vertices. Two of these are maximal independent sets.

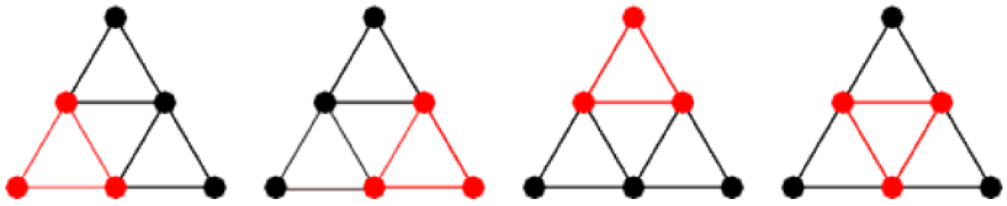


Cliques

A **clique** is a subset of vertices of an undirected graph G such that every two distinct vertices in the clique are adjacent; that is, its induced subgraph is complete. The task of finding whether there is a clique of a given size in a graph (the clique problem) is NP-complete, but despite this hardness result, many algorithms for finding cliques have been studied.

A **maximal clique** is a clique that cannot be extended by including one more adjacent vertex, that is, a clique that does not exist exclusively within the vertex set of a larger clique.

A **maximum clique** of a graph, G, is a clique, such that there is no clique with more vertices. Moreover, the **clique number** ω(G) of a graph G is the number of vertices in a maximum clique in G.



Bron-Kerbosch algorithm

The Bron-Kerbosch algorithm is an efficient method for finding all maximal cliques in a graph. It was designed by Dutch scientists Coenraad Bron and Joep Kerbosch, who published its description in 1973. Tomita et al. (2006) gave a depth-first search algorithm with pruning methods that are similar to the Bron-Kerbosch algorithm.

The basic form of the Bron–Kerbosch algorithm is a recursive backtracking algorithm that searches for all maximal cliques in a given graph G. More generally, given three disjoint sets of vertices R, P, and X, it finds the maximal cliques that include all of the vertices in R, some of the vertices in P, and none of the vertices in X.

In each call to the algorithm, P and X are disjoint sets whose union consists of those vertices that form cliques when added to R. In other words, P ∪ X is the set of vertices that are joined to every element of R. When P and X are both empty there are no further elements that can be added to R, so R is a maximal clique and the algorithm outputs R.

The recursion is initiated by setting R and X to be the empty set and P to be the vertex set of the graph. Within each recursive call, the algorithm considers the vertices in P in turn; if there are no such vertices, it either reports R as a maximal clique (if X is empty) or backtracks. For each vertex v chosen from P, it makes a recursive call in which v is added to R and in which P and X are restricted to the neighbour set N(v) of v, which finds and reports all clique extensions of R that contain v. Then, it moves v from P to X to exclude it from consideration in future cliques and continues with the next vertex in P.

That is, in pseudocode, the algorithm performs the following steps:

```
algorithm BronKerbosch1(R, P, X) is
    if P and X are both empty then
        report R as a maximal clique
    for each vertex v in P do
        BronKerbosch1(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

APPLICATIONS OF GRAPH COLORING

MAKING SCHEDULES OR TIMETABLES

Exam schedules need to be in such a way that no two exams with a common student are scheduled at the same time. This problem can be represented as a graph where every vertex is a subject and an edge between two vertices means there is a common student. So this is a graph coloring problem where the minimum number of time slots is equal to the chromatic number of the graph.

MOBILE RADIO FREQUENCY ASSIGNMENT

When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. This problem is also an instance of a graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in the range of each other.

REGISTER ALLOCATION

In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.

MAP COLORING

Geographical maps of countries or states where no two adjacent cities cannot be assigned the same color. Four colors are sufficient to color any map.

SUDOKU

Sudoku is also a variation of the Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in the same row or same column or same block.

LITERATURE REVIEW

A List based Approach to Solve Graph Coloring Problem

Authors: Ajay Narayan Shukl, M.L. Garg

Publication: 2018 International Conference on System Modeling & Advancement in Research Trends (SMART), IEEE

Overview:

It proposes a list based algorithm:

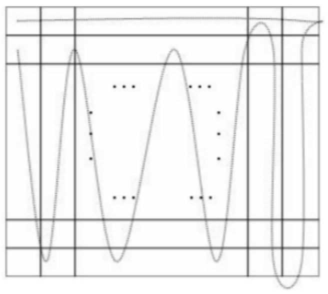
- It starts with accessing the adjacency list from top to bottom and reads every vertex of the graph with the help of a down pointer.
- If it is the first/start vertex of the graph then this vertex may be assigned any colour from the colour list.
- Otherwise, appropriate colour is retrieved from the colour list & assigned to vertices in the graph.
- After assigning a colour to a vertex in the graph remove the assigned colour from the colour list of all adjacent vertices of the vertex which has assigned the colour.

A Study of the Sudoku Graph Family

Authors: Hilmi Yildirim; M. S. Krishnamoorthy; Narsingh Deo

Overview:

A Sudoku puzzle of order B has B^2xB^2cells which are to be filled by the numbers from 1 to B^2. It has the following properties:



Hamiltonian cycles in generalized Sudoku Graphs

- G_B has B^4 vertices and $\frac{B^4(B-1)(3B+1)}{2}$ edges.
- $G_B$is $(B - 1)(3B + 1)$-regular (G_3 is 20-regular).
- All Sudoku graphs are Hamiltonian.
- $G_B$is Eulerian if and only if B is odd.
- Diameter(G_B) = Radius(G_B) = 2
- Clique Number, $\omega(G_B)$ = B^2since there are 3B^2cliques of size B^2in G_B.
- $G_B$is B-partite and B-colorable by definition.

Graph Coloring with Minimum Colors: An Easy Approach

Authors: Amit Mittal; Parash Jain; Srushti Mathur; Preksha Bhatt

Publication: 2011 International Conference on Communication Systems and Network Technologies, IEEE

Overview:

It proposes the following algorithm that colours the vertices using a minimum number of colours. First, you make a table of all the edges in the graph. Then check each edge one by one:

- If both vertices of the edge are of different colours, no changes are made.
- If they have the same colour, then the latter of the vertices is assigned a colour value greater than the first vertex.

Finding All Cliques of an Undirected Graph

Authors: Coen Bron and Joep Kerboscht

Overview:

The Bron-Kerbosch algorithm is one of the fastest algorithms to find all maximal cliques. The Bron–Kerbosch algorithm is an enumeration algorithm for finding maximal cliques in an undirected graph. That is, it lists all subsets of vertices with the two properties that each pair of vertices in one of the listed subsets is connected by an edge, and no listed subset can have any additional vertices added to it while preserving its complete connectivity.

Math and Sudoku: Exploring Sudoku Boards Through Graph Theory, Group Theory, and Combinatorics

Authors: Kyle Oddson

Overview:

In this paper, the author has generalized a chromatic polynomial for the sudoku graph. He has done the total enumeration of a 4x4 Sudoku board and proved that there exist 288 total 4x4 sudoku boards. He has then researched diagonal 4x4 Sudoku Boards (those with the added constraint that each of the main diagonals must also contain the numbers {1, 2, 3, 4}). He has then proved that there exists one fundamentally distinct diagonal 4x4 sudoku board.

METHODOLOGY

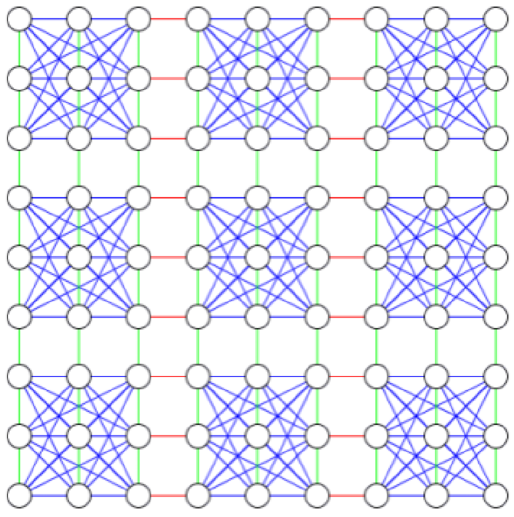
INITIALIZING THE SUDOKU GRAPH

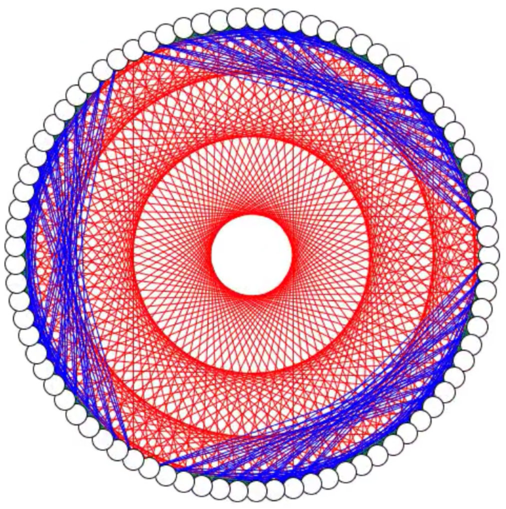
A typical sudoku graph consists of 9x9 boxes (i.e. 81 vertices). The sudoku puzzle can be visualized as a graph with 81 vertices where each small box in the Sudoku grid represents a vertex.

After creating the vertices we add the edges between these vertices. We can see that there are a total of 810 edges. Each vertex has the following edge:

- An edge with all the vertices in its respective 3x3 grid.
- An edge with all the vertices in its respective row.
- An edge with all the vertices in its respective column.

Therefore each vertex in a Sudoku graph has a total of 20 edges.





SOLUTION OF THE SUDOKU PUZZLE

We are applying the concept of the maximum independent set to solve the partially colored Sudoku puzzle. By definition, the maximum independent set in a graph is a subset of vertices that are not connected in pairs by edges. In a completed sudoku, we can conclude that all vertices which are filled by the number 1 have no edges in common. Similarly, the entire graph contains 9 such independent sets, one for each number. Therefore if we find the maximum independent set for a sudoku puzzle, we can optimize vertex coloring.

FINDING MAXIMUM INDEPENDENT SET USING MAXIMUM CLIQUES

Finding the maximum independent set of any graph is an NP-Hard problem. Hence in order to find a more feasible solution, we apply the following theorem instead.

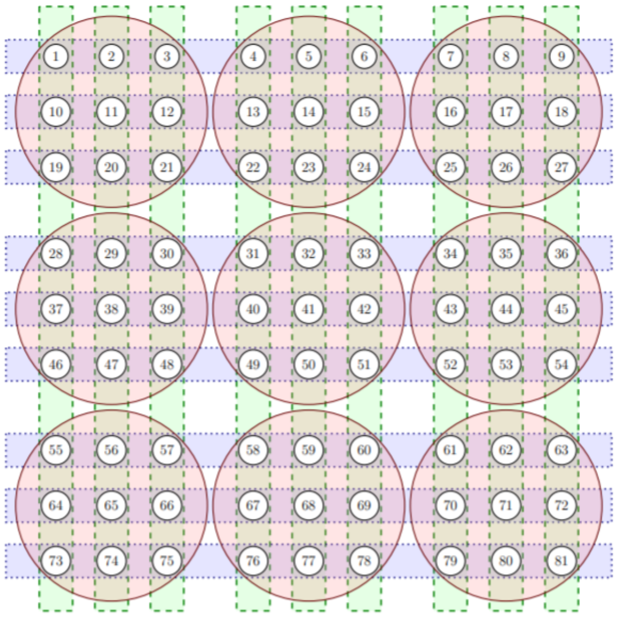
The maximum independent set in a graph is the maximum clique in a complement graph.

The complement graph of any graph is a graph such that it contains only those edges which are not present in the original graph.

Therefore, in order to find the maximum independent set of the graph, we can just take the complement of the sudoku graph and find its maximum clique instead as the algorithms to find all maximal cliques are a lot faster.

FINDING MAXIMAL CLIQUES USING BRON KERBOSCH

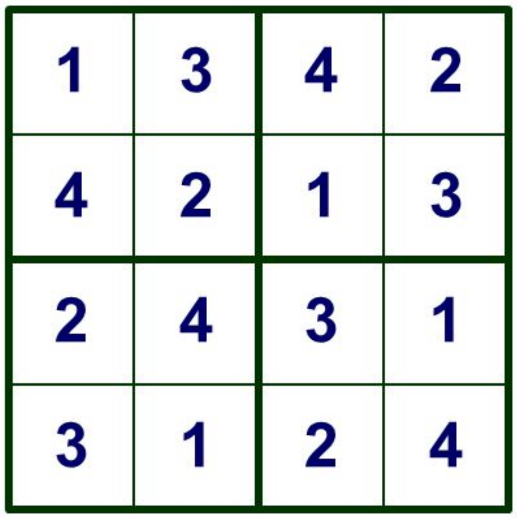
The Bron Kerbosch algorithm defined previously is used to find all the maximal cliques present in the sudoku graph. The maximum clique is then determined from this list of maximal cliques. The clique with the highest number of vertices i.e the clique with maximum cardinality is the maximum clique.



Graph formulation of a Sudoku with maximal cliques highlighted

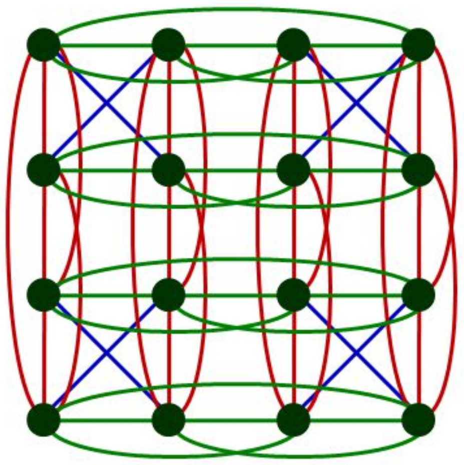
SOLVING 4x4 SUDOKU USING MAXIMUM INDEPENDENT SET

Solving the sudoku by using the maximum independent set rather than simply brute forcing a solution is a more elegant solution. However, the trick we apply, where we find the maximum clique of the complement graph, causes the time complexity to be very high. Hence, for the sake of simplicity, we solve a simple 4x4 sudoku puzzle instead. The graph for this sudoku, G looks like this:



1	3	4	2
4	2	1	3
2	4	3	1
3	1	2	4

4 X 4 Sudoku
16 nodes



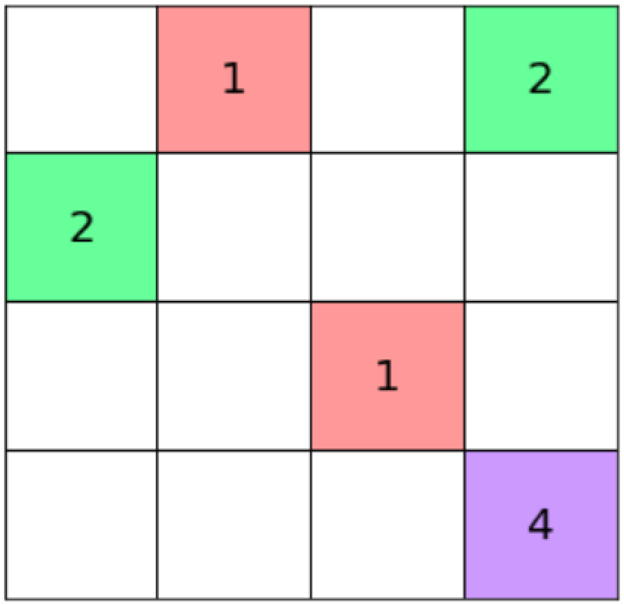
All constraints
56 edges

Then the steps are simple:

1. We find the complement of this graph, let say, H.
2. We find all maximal cliques for H. The clique with the maximum cardinality is the maximum clique, C.
3. This graph, C, is also the maximum independent set of G.
4. We extract the solution of the sudoku, i.e colour the graph in accordance to the edges in C.
5. We output the graph.

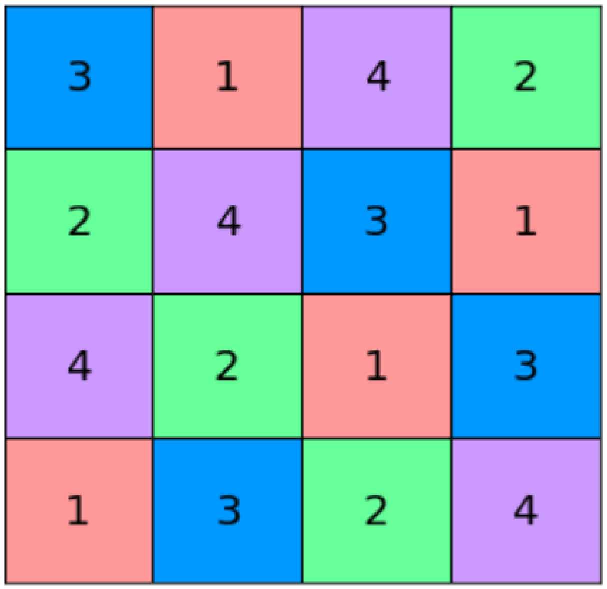
RESULTS

INPUT (UNSOLVED SUDOKU AS GRAPH)



	1		2
2			
		1	
			4

OUTPUT OBTAINED (SOLVED SUDOKU WITH GRAPH COLORING)



3	1	4	2
2	4	3	1
4	2	1	3
1	3	2	4

BIBLIOGRAPHY

1. https://www.geeksforgeeks.org/graph-coloring-applications/

2. https://ieeexplore.ieee.org/document/8746966

3. https://www.researchgate.net/publication/264919460_A_study_of_the_Sudoku_graph_family

4. https://www.ics.uci.edu/~johnsong/papers/Bron%20and%20Kerbosch%20-%20Finding%20All%20Cliques%20of%20an%20Undirected%20Graph.pdf

5. https://ieeexplore.ieee.org/document/5966527

6. https://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=1078&context=studentsymposium

7. http://pi.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Sudokus_as_Graphs.html

8. https://medium.com/code-science/sudoku-solver-graph-coloring-8f1b4df47072

APPENDIX - I

Solving Sudoku: Application of Graph Colouring

Code:

```
import numpy as np
import os
import matplotlib.pyplot as plt
import·matplotlib.patches·as·patches
```

```
import random as rd
import copy
```

```
''' A class that defines all all the functions for a graph data structure'''
class graph:

    def __init__(self):
        self.graph={}
        self.visited={}

    def append(self,vertexid,edge,weight):
        if vertexid not in self.graph.keys():
            self.graph[vertexid]={}
            self.visited[vertexid]=0
        if edge not in self.graph.keys():
            self.graph[edge]={}
            self.visited[edge]=0
        self.graph[vertexid][edge]=weight

    def reveal(self):
        return self.graph

    def vertex(self):
        return list(self.graph.keys())

    def edge(self,vertexid):
        return list(self.graph[vertexid].keys())

    def edge_reverse(self,vertexid):
        return [i for i in self.graph if vertexid in self.graph[i]]

    def weight(self,vertexid,edge):
        return (self.graph[vertexid][edge])
    def order(self):
        return len(self.graph)

    def visit(self,vertexid):
        self.visited[vertexid]=1

    def go(self,vertexid):
        return self.visited[vertexid]

    def route(self):
        return self.visited

    def degree(self,vertexid):
        return len(self.graph[vertexid])

    def mat(self):
        self.matrix=[[0 for _ in range(max(self.graph.keys())+1)] for i in range(max(self.gra
        for i in self.graph:
            for j in self.graph[i].keys():
                self.matrix[i][j]=1
        return self.matrix

    def remove(self,vertexid):
        for i in self.graph[vertexid].keys():
            self.graph[i].pop(vertexid)
        self.graph.pop(vertexid)

    def disconnect(self,vertexid,edge):
        del self.graph[vertexid][edge]

    def clear(self,vertexid=None,whole=False):
        if whole:
            self.visited=dict(zip(self.graph.keys(),[0 for i in range(len(self.graph))]))
        elif vertexid:
            self.visited[vertexid]=0
        else:
            assert False,"arguments must satisfy whole=True or vertexid=int number"
```

Bron-Kerbosch Algorithm

```
def bron_kerbosch(ADT,R=set(),P=set(),X=set()):
    """Bron Kerbosch algorithm to find maximal cliques
        P stands for priority queue, where pending vertices are
        R stands for result set, X stands for checked list
        To find maximal cliques, only P is required to be filled
        P=set(ADT.vertex())"""
    if not P and not X:
        yield R
    while P:
        v=P.pop()
        yield from bron_kerbosch(ADT,R=R.union([v]),P=P.intersection(ADT.edge(v)),X=X.interse
        X.add(v)
```

Functions to create and solve the Sudoku

```
def create_graph_structure(num):

    ADT=graph()

    #locate the coordinates of submatrices
    pivots=[i for i in range(0,num,int(num**0.5))]
    intervals=[range(i,i+int(num**0.5)) for i in pivots]
    submatrices=[(i,j) for i in intervals for j in intervals]

    #connect elements in the same submatrix
    for submatrix in submatrices:
        vertices=[(i,j) for i in submatrix[0] for j in submatrix[1]]
        for i in range(len(vertices)):
            for j in range(i+1,len(vertices)):
                ADT.append(vertices[i],vertices[j],0)
                ADT.append(vertices[j],vertices[i],0)

    #connect elements in the same column
    for latitude in range(num):
        cols=[(latitude,i) for i in range(num)]
        for i in range(len(cols)):
            for j in range(i+1,len(cols)):
                ADT.append(cols[i],cols[j],0)
                ADT.append(cols[j],cols[i],0)

    #connect elements in the same row
    for longitude in range(num):
        rows=[(i,longitude) for i in range(num)]
        for i in range(len(rows)):
            for j in range(i+1,len(rows)):
                ADT.append(rows[i],rows[j],0)
                ADT.append(rows[j],rows[i],0)

    return ADT
```

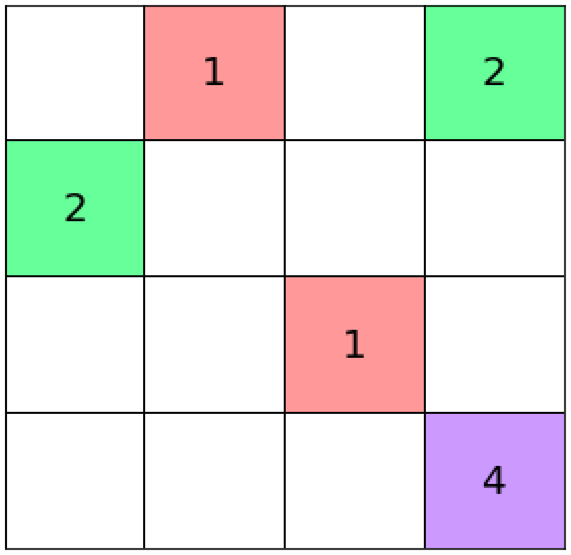
```
def get_sudoku_graph(puzzle):

    #convert sudoku to graph
    num=puzzle.shape[0]
    ADT=create_graph_structure(num)
    exclude_vertices=[(np.where(puzzle!=0)[0][i],np.where(puzzle!=0)[1][i]) for i in range(le
    coloring_graph=graph()
    for node in ADT.vertex():

        #create clones to form cliques
        for from_color in range(1,num+1):
            for to_color in range(from_color+1,num+1):
                coloring_graph.append((node[0],node[1],from_color),
                                      (node[0],node[1],to_color),0)
                coloring_graph.append((node[0],node[1],to_color),
                                      (node[0],node[1],from_color),0)
        #connect clones
        for link in ADT.edge(node):
            for color in range(1,num+1):
                coloring_graph.append((node[0],node[1],color),
                                      (link[0],link[1],color),0)
                coloring_graph.append((link[0],link[1],color),
                                      (node[0],node[1],color),0)
    #remove filled cells
    for node in exclude_vertices:
        current_vertex=(node[0],node[1],puzzle[node])
        for neighbor in coloring_graph.edge(current_vertex):
            coloring_graph.remove(neighbor)
        coloring_graph.remove(current_vertex)
    return coloring_graph
```

```
def extract_solution(puzzle,ans):

    #initialize
    solution=puzzle.copy()

    #revert flatten matrix to real sudoku
    for i in ans:
        solution[(i[0],i[1])]=i[2]

    return solution
```

```
def viz(sudoku,colors):

    ax=plt.figure(figsize=(5,5)).add_subplot(111)

    #plot blocks with graph coloring
    for latitude in range(sudoku.shape[0]):
        for longitude in range(sudoku.shape[1]):
            ax.add_patch(patches.Rectangle((latitude,longitude),
                    1,1,facecolor=colors[sudoku[latitude][longitude]],
                                           edgecolor='k',fill=True))

    #show answers
    for latitude in range(sudoku.shape[0]):
        for longitude in range(sudoku.shape[1]):
            if sudoku[latitude][longitude]!=0:
                plt.text(latitude+0.5,longitude+0.5,
                     str(sudoku[latitude][longitude]),
                     ha='center',va='center',fontsize=20)

    plt.xlim((0,sudoku.shape[0]))
    plt.ylim((0,sudoku.shape[1]))
    plt.xticks([])
    plt.yticks([])
    plt.show()
```

Solving a 4x4 Sudoku Graph Puzzle

```
puzzle=np.array([
[0,0,2,0],
[0,0,0,1],
[0,1,0,0],
[4,0,0,2]])
viz(puzzle,{0: '#ffffff',1:'#ff9999',2:'#66ff99',3:'#0099ff',4:'#cc99ff'})
```



```
coloring_graph=get_sudoku_graph(puzzle)
```

```
complement_graph=graph()
for node in coloring_graph.vertex():
    connections=[link for link in coloring_graph.vertex() if link not in coloring_graph.edge(
    for link in connections:
        complement_graph.append(node,link,0)
```

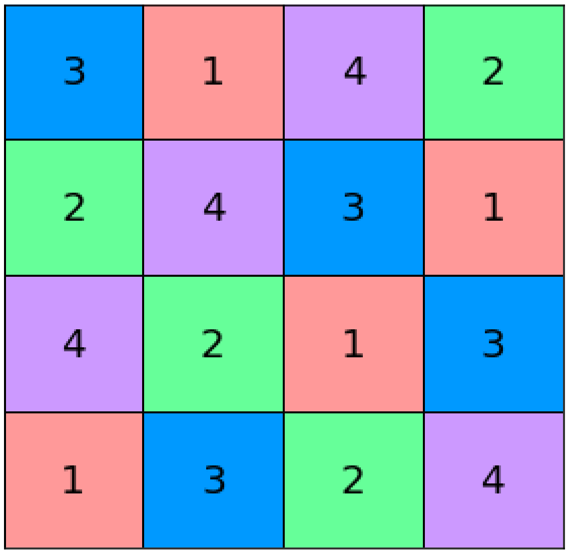
```
mis=list(bron_kerbosch(complement_graph, P=set(complement_graph.vertex())))
```

```
ans=max(mis,key=len)
```

```
sudoku=extract_solution(puzzle,ans)
sudoku
```

```
array([[1, 4, 2, 3],
       [3, 2, 4, 1],
       [2, 1, 3, 4],
       [4, 3, 1, 2]])
```

```
#viz
viz(sudoku,{1:'#ff9999',2:'#66ff99',3:'#0099ff',4:'#cc99ff'})
```



Submitted By:

Aiman Siddiqua 2K18/MC/008

Apoorva 2K18/MC/019