Ho Chi Minh University
Of Technology

Discrete Math

CO1006

SUBGRAPH ISOMORPHISM

IDENTIFYING A GIVEN SUB-GRAPH IN A LARGER GRAPH

Group: CO1006_ TD6

Member:

Students	ID
Trương Ngọc Khánh	2211538
Nguyễn Duy Khiêm	2211571
Bùi Đăng Khoa	2211581
Lê Đăng Khoa	2211601
Phan Huy Trung	2213709

December, 2023

Table of Contents

Static Graph Challenge: Subgraph Isomorphism **1**

Chapter I. Introduction . . . 1
Chapter II. Preliminaries . . . 2
TRIANGLES . . . 2
K-TRUSS . . . 2
INCIDENCE MATRIX . . . 2
ADJENCE MATRIX . . . 4
DEGREE . . . 5
GRAPH ISOMORPHISM . . . 6
Chapter III. Approaches . . . 6
Triangles Counting . . . 6
K truss . . . 7
Chapter IV. EXPERIMENT AND ANALASYS . . . 8
FINDING THE MAX KTRUSS . . . 9
TRIANGLES COUNTING . . . 12
Chapter V. Conclusion . . . 14
References . . . 15

Describe the Chinese postman problem and explain how to solve this problem. **15**

Chapter I. Introduction . . . 17
Chaptere II. Solving the Chinese Postman Problem . . . 17
Chapter III. Conclusion . . . 18
References . . . 19

Discuss the history of The Four Color Theorem **19**

Chapter I. Introduction . . . 21
Chapter II. Mathematical foundations and history . . . 21
Chapter III. Conclusion . . . 24
References . . . 24

Static Graph Challenge: Subgraph Isomorphism

Abstract

Subgraph isomorphism plays a crucial role in pattern matching and finding structural similarities within complex networks. The proposed approach leverages the triangle enumeration algorithm to identify the presence and quantity of triangles within the graph, shedding light on the intricate relationships between vertices. Furthermore, the study delves into the identification of k-trusses, a structural concept capturing cohesive subgraphs within the larger network. The enumeration of triangles and the identification of k-trusses contribute to a comprehensive understanding of the graph's structural complexity and connectivity patterns. These insights are invaluable in various domains, including social networks, biological systems, and recommendation engines. In this article, we will rely on triangles and trusses to analyze a data set in depth. This study is based on Discrete and Mathematical assignments.

Chapter I. Introduction

Graphs, as versatile mathematical structures, provide a powerful framework for modeling relationships and dependencies in diverse fields ranging from computer science and biology to social networks and logistics. The intrinsic value of graphs lies in their ability to represent complex systems in a way that captures the essence of connections between entities. This has spurred a wealth of applications aimed at extracting meaningful insights, identifying patterns, and solving complex problems.

One particularly challenging problem that arises in the realm of graph theory is the subgraph isomorphism challenge. The motivation to address this problem is rooted in the need to decipher intricate structures within larger networks or datasets. At its core, the subgraph isomorphism problem involves determining whether a specific pattern, represented by a smaller graph, can be found within a larger graph. This problem encapsulates a wide array of real-world scenarios where recognizing and understanding the presence of certain configurations or relationships is paramount. The subgraph isomorphism problem extends across various domains. For example: in chemistry, subgraph isomorphism aids in identifying chemical structures within vast databases, facilitating drug discovery by pinpointing potential compounds with desired properties, in cybersecurity, solving the subgraph isomorphism problem is essential for identifying anomalous patterns or specific attack structures, enhancing the robustness of cybersecurity measures, etc ...

Subgraph isomorphism is an NP-complete problem, but its smaller components can be solved in polynomial time. Therefore, to optimize the problem, we look for algorithms with polynomial time complexity in this paper. Two algorithms introduced in this paper are triangle counter and finding the max k-truss. A triangle is a special case of subgraph isomorphism where the subgraph in question is a triangle. However, triangles are not specialized and are not a solution for the general subgraph; they are not designed to find arbitrary subgraphs other than triangles. Nevertheless, both triangle counting and k-truss finding algorithms are part of a larger algorithm to solve the general problem. These algorithms help reduce the search space to perform faster and less complex algorithms.

GraphChallenge.org provides a wide range of pre-parsed graph data sets, which are graphs with edge counts that can range in the millions or billions. The Graph Challenge is designed to work on arbitrary graphs drawn from both real-world data sets and simulated data sets. Examples of real-world data sets include the Stanford Large Network Dataset Collection (see ttp://snap.stanford.edu/data), the AWS Public Data Sets (see aws.amazon.com/publicdata-sets), and the Yahoo! Webscope Datasets (see webscope.sandbox.yahoo.com). These real-world data sets cover a wide range of applications and data sizes.

In this research paper, we have chosen the dataset Arxiv GR-QC (General Relativity and Quantum Cosmology) taken from SNAP. This collaboration network is from the e-print arXiv and covers scientific collaborations between authors papers submitted to General Relativity and Quantum Cosmology category. The dataset demonstrates that: If an author i co-authored a paper with author j, the graph contains a undirected edge from i to j. If the paper is co-authored by k authors, this generates a completely connected (sub)graph on k nodes. The data covers papers in the period from January 1993 to April 2003 (124 months). It begins within a few months of the inception of the arXiv, and thus represents essentially the complete history of its GR-QC section. This dataset has 5242 nodes and 14484 edges.

In each case, the data has been parsed so that all vertices are integers from 1 to the total number of vertices n in the graphs and all edges weights are set to a value of 1. The dataset provided in two formats TSV file and MMIO file.

The format of adjency files is shown here, with the starting and ending vertex be store in the m elements vector u and v:

$$\begin{array}{ccc} \mathbf{u(1)} & \mathbf{v(1)} & 1 \\ . & . & . \\ . & . & . \\ . & . & . \\ \mathbf{u(m)} & \mathbf{v(m)}) & 1 \end{array}$$

$$\text{where } i = 1, ..., m,\ \mathbf{u(i)} \in \{1, ..., n\},\ \text{ and } \mathbf{v(i)} \in \{1, ..., n\}$$

The format of incidence files is shown here, with the egde be store in m elements vector e and associated vertex be store in n elements vector v.

$$\begin{array}{rcc} \mathbf{v}(i_1) & \mathbf{e(1)} & 1 \\ \mathbf{v}(i_2) & \mathbf{e(1)} & 1 \\ . & . & . \\ . & . & . \\ . & . & . \\ \mathbf{v}(i_{2m-1}) & \mathbf{e(m)} & 1 \\ \mathbf{v}(i_{2m}) & \mathbf{e(m)} & 1 \end{array}$$

$$\text{where } i = 1, ..., m,\ \mathbf{e(i)} \in \{1, ..., m\},\ \text{ and } x = 1, ..., 2m, \mathbf{v(i)} \in \{1, ..., n\}$$

Filtering on edge/vertex labels is often used when available to reduce the search space of many graph analytics.

In this article, we will try to analyze the network of co-authorship groups from low to high levels of connection in the graph. These analyzes are based on the algorithms listed in section III.

Chapter II. Preliminaries

TRIANGLES

Triangles are the most basic trivial sub-graph. A triangle can be defined as a set of three mutually adjacent vertices in a graph.

A triangle count in a graph is the number of sets of three vertices that are connected by edges, forming a triangle shape. It is a measure of how densely connected a graph is, and how likely it is that nodes form communities or clusters.

K-TRUSS

A graph's k-truss is the maximal induced subgraph in which all edges are incident to at least $k - 2$ triangles within the subgraph. A k-truss is basically a subgraph composed solely of triangles, which provides significant stability.

As the number k increases, fewer of the graph's components remain in the resulting subgraph, leaving only the nodes and edges that form the most relevant structures of the graph.

Figure: 3-truss subgraph in the larger graph

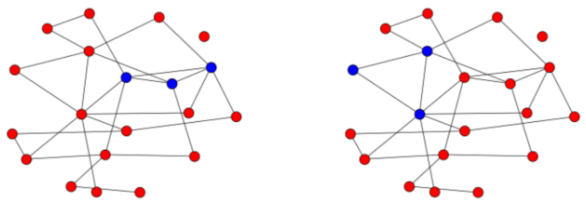
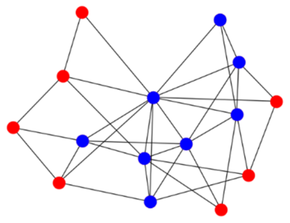


Figure: 4-truss subgraph in the larger graph



INCIDENCE MATRIX

In mathematics, an incidence matrix is a logical matrix that shows the relationship between two classes of objects, which are vertices and edges. It is also a two-dimensional array $(|V| \times |E|)$ with:

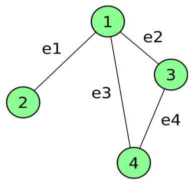
+ The rows of an incidence matrix show the number of vertices.

+ The columns of an incidence matrix show the number of edges.

Incidence Matrix can be considered under two cases of graphs: Undirected Graphs and Directed Graphs.

Undirected Graphs: $M_{ij} = \begin{cases} 1 \text{ if } v_i \text{ is incident with } e_j \\ 0 \text{ otherwise} \end{cases}$

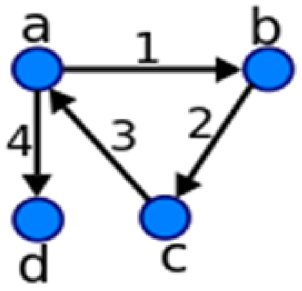
For example:



	e_1	e_2	e_3	e_4
1	1	1	1	0
2	1	0	0	0
3	0	1	0	1
4	0	0	1	1

Directed Graphs:$M_{ij} = \begin{cases} 1 \text{ if the vertex } v_i \text{ has an outgoing edge } e_j \\ -1 \text{ if the vertex } v_i \text{ has an incoming edge } e_j \\ 0 \text{ otherwise} \end{cases}$

For example:



	e_1	e_2	e_3	e_4
a	1	0	-1	1
b	-1	1	0	0
c	0	-1	1	0
d	0	0	0	-1

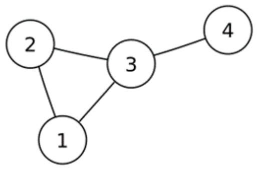
ADJENCE MATRIX

The adjacency matrix, also called the connection matrix, is a matrix containing rows and columns which is used to represent a simple labelled graph, with 0 or 1 in the position of (v_i, v_j) according to the condition whether v_i and v_j are adjacent or not. It is a 2D dimension $(|V| \times |V|)$.

Adjacence Matrix can be considered under two cases of graphs: Undirected Graphs and Directed Graphs.

Undirected Graphs: $M_{ij} = \begin{cases} 1 \text{ if } v_i \text{ is adjacent to } v_j. \\ 0 \text{ otherwise.} \end{cases}$

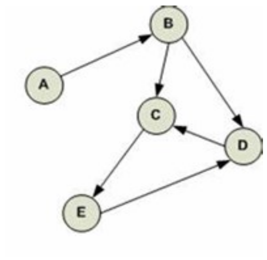
For example:



	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

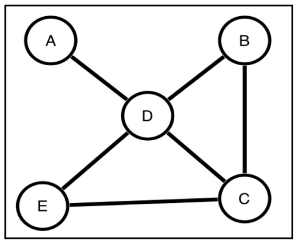
Directed Graphs:$M_{ij} = \begin{cases} 1 \text{ if indicates an edge from i to j.} \\ 0 \text{ otherwise.} \end{cases}$

For example:



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	0	0	1	0	0
E	0	0	0	1	0

DEGREE

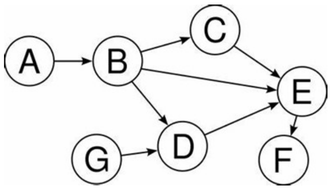


Degree of a vertex is the number of edges that are incident to that vertex, denoted as $deg(v)$.

In an undirected graph, for example:
deg(A) = 1
deg(B) = 2
deg(C) = 3
deg(D) = 4
deg(E) = 2

In a directed graph:

- In-degree of a vertex v, denoted by $deg^{-}(v)$, is the number of edges with v as their terminal vertex.
- Out-degree of a vertex v, denoted by $deg^{+}(v)$, is the number of edges with v as their initial vertex.

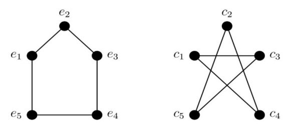


Out Degree	In Degree
$deg^{+}(A) = 1$	$deg^{-}(A) = 0$
$deg^{+}(B) = 3$	$deg^{-}(B) = 1$
$deg^{+}(C) = 1$	$deg^{-}(C) = 1$
$deg^{+}(D) = 1$	$deg^{-}(D) = 2$
$deg^{+}(E) = 1$	$deg^{-}(E) = 3$
$deg^{+}(F) = 0$	$deg^{-}(F) = 1$
$deg^{+}(G) = 1$	$deg^{-}(G) = 0$

GRAPH ISOMORPHISM

The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$
are isomorphic if there exists a one-to-one and onto function f from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if f(a) and f(b) are adjacent in G_2, for all a and b in V_1. Such a function f is called an isomorphism. In other words, when two simple graphs are isomorphic, there is a one-to-one correspondence between vertices of the two graphs that preserves the adjacency relationship.

If an isomorphism exists between two graphs, then the graphs are called isomorphic and denoted as: $G_1 \cong G_2$.



$f(e_1) = c_1$
$f(e_2) = c_3$
$f(e_3) = c_5$
$f(e_4) = c_2$
$f(e_5) = c_4$

Chapter III. Approaches

Triangles Counting

```
Algorithm 1: Array based implementation of triangle
counting algorithm using the adjacency and incidence
matrix of a graph
Data: Adjacency matrix A and incidence matrix E
Result: Number of triangles in graph G
initialization:
C=AE
$n_T = nnz(C)/3$

Multiplication is overloaded such that
C(i,j) = {i,x,y} iff
A(i,x) = A(i,y) = 1 & E(x,j) = E(y,j) = 1
```

This algorithm proposed by Wolf. Use an overloaded matrix multipication approach on the adjacency and incidence matrices of the graph.

Input Data: Adjacency matrix **A** and incidence matrix **E**. Because the received dataset is a directed graph, the transformation is needed to make the matrix **A** symmetrical:

$$\mathbf{A} = \mathbf{A} + \mathbf{A}^T$$

From the definition, we can get the triangle based on the association between the vertex and edges.

In the matrix **E**, each column corresponding the edge has two entries with value equal to 1 corresponding two endpoints of this edge. A triangle appears when there is an intersection between two adjacent sets of vertices between the two endpoints. If on the row of the matrix **A** corresponds to a vertex of the graph that is associated with the two endpoints. That means between those two endpoints there exists at least one common adjacent vertex and there is a triangle created from those vertices.Repeat this over each edge and vertex of the graph. This is equivalent to multiplying each row of matrix **A** by each column of matrix **E**.

Therefore we use matrix multiplication between **A** and **E**:

$$\mathbf{C{=}A.E}$$

The value we need to find for each entry in the matrix **C** is 2. With the entry **C**[i,j] that has the value equal to 2, this means there are 2 edges connecting edge j to vertex i. To optimize memory, apply to each entry the function that 2 maps 1 and all other values to 0.

Counting the number of triangles depends on the number of non-zero values in matrix **C**: nnz(C)

Each entry in matrix **C** represents a vertex and a an edge. A triangle is made up of 3 vertices and 3 edges, so triangle counting is calculated as follows:

$$n_T = nnz(C)/3$$

Define the triangle as follows:

$$\mathbf{C}[i,j] = i, x, y \text{ iff}$$

$$\mathbf{A}(i,x) = \mathbf{A}(i,y) = 1 \ \& \ \mathbf{E}(x,j) = \mathbf{E}(y,j) = 1$$

The two endpoints x, y of side j both have an adjacent vertex i, which forms a triangle {i,x,y}

K truss

Algorithm 4: Array based implementation of K-Truss algorithm.

```
Data: Unoriented incidence matrix $\mathbf{E}$ and integer k.
Result: Incidence matrix of k-truss subgraph $\mathbf{E}_k$
initialization:
$d_x = sum(\mathbf{E}_x)$
$\mathbf{A} = \mathbf{E}^T\mathbf{E} - diag(d)$
$\mathbf{R{=}EA}$
$s = (\mathbf{R} == 2)\mathbf{1}$
$x = find(s < k - 2)$
while x is not empty do
    $\mathbf{E}_x = \mathbf{E}(x, :)$
    $\mathbf{E} = \mathbf{E}(x_c, :)$
    $d_x = sum(\mathbf{E}_x)$
    $\mathbf{R} = \mathbf{R}(x_c, :)$
    $\mathbf{R} = \mathbf{R} - \mathbf{E}(\mathbf{E}_x^T.\mathbf{E}_x - diag(d_x))$
    $s = (\mathbf{R} == 2)\mathbf{1}$
    $x = find(s < k - 2)$
end
```

Input data: Unoriented incidence matrix **E** and integer k.

Each row of **E** has a 1 in the column of each associated vertex (or the other words, the two endpoints of the edge). To find the triangle, we need to find the intersection of adjacent vertices of these vertex. If the columns of the adjacency matrix **A** associated with the two vertices that need to be considered, this means that for two endpoints of the edge, there exists at least one common adjacent vertex. This is the condition for a triangle. Repeat that with other edges and vertices, we get a matrix multiplication:

$$\mathbf{R}{=}\mathbf{E.A}$$

Similarly to triangle counting algorithm, we apply to each entry the function that 2 maps 1 and all other values to 0 and count the number of 1 values in each row to store in row indices s.

$$s = (\mathbf{R} == 2)\mathbf{1}$$

Note: Because the input data is incidence matrix **E**, there needs to be a transition from **E** to **A**.

$$\mathbf{A} = \mathbf{E}^T\mathbf{E} - diag(\mathbf{E}^T\mathbf{E}).$$

$diag(\mathbf{E}^T\mathbf{E})$: diagonal matrix with the value of each entry on the diagonal being the degree of the vertex corresponding to that entry.

The degree of the vertex can be calculate by the function:

sum(**E**): return the vector that containing the sum of each column of **E**.

$$\Rightarrow \mathbf{A} = \mathbf{E}^T\mathbf{E} - diag(d) \text{ with d = sum(d)}$$

$$x = find(s < k - 2)$$

$find$ function: return the linear indices corresponding to the value of which index in s is less than $k-2$ in the set of row indices.

This follow from the definition, where s is the set of row indices that count the associated vertex to each edge in matrix **R**. To determine k truss, it is necessary to remove edges that are not inside or are inside but less than $k-2$ triangles. Repeat deleting those edges until all edges satisfy the k truss condition.

Note: After removing those edges , the degrees at two vertices of the edge will decrease. Therefore, both matrices A and E have changed, requiring **A** and **E** to be updated.

Update matrix **E** with x_c is the complement of x:

$$\mathbf{E} = \mathbf{E}(x_c, :).$$

Update the matrix **R**, because degrees of the associated of the associated vertex with the dropped edge will be reduced.

$$\mathbf{E}_x = \mathbf{E}(x, :)$$
$$d_x = sum(\mathbf{E}_x)$$

Delete the row x_c in **R**:

$$\mathbf{R} = \mathbf{R}(x_c, :)$$

Update R:

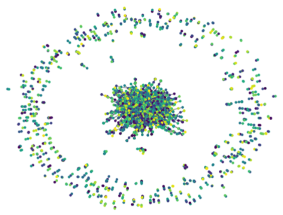
$$\mathbf{R} = \mathbf{R} - \mathbf{E}(\mathbf{E}_x^T.\mathbf{E}_x - diag(d_x))$$

Continue repeat until set x is empty.

Chapter IV. EXPERIMENT AND ANALASYS

The results obtained from Google Colab: GPU Tesla K80 with 12GB VRAM and computational power of 15.7 TFLOPs. All code is written in the Python programming language.

Visualize the dataset graph :



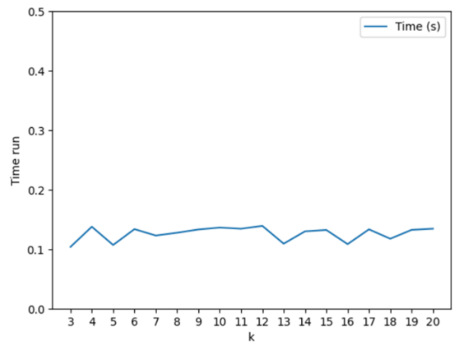
FINDING THE MAX KTRUSS

Code function for the algorithm:

```
import numpy as np
import scipy as sp
from scipy.sparse import csr_matrix
from scipy.sparse import csc_matrix
from scipy.sparse import coo_matrix
from scipy.sparse import lil_matrix
import time
import pandas as pd
import scipy.io as sio

##################################################
##################################################
def StrArrayRead(filename):

f=open(filename,'r')
edgelist = []
with open(filename, 'r') as f:
for line in f:
edgelist.append(list(map(float, line.split('\t'))))
f.close()
return np.asarray(edgelist)

##################################################
##################################################
def set_zero_rows(sparray, rowNum):
for row in rowNum:
sparray.data[sparray.indptr[row]:sparray.indptr[row+1]]=0

##################################################
##################################################
def set_diag_val(sparray,val):
r,c=sparray.shape
for row in range(r):
sparray[row,row]=val

##################################################
```

```
##################################################

def ktruss (inc_mat_file,k):

ii=StrArrayRead(inc_mat_file)

startTime=time.perf_counter()

E=csr_matrix(( ii[:,2], (ii[:,0]-1, ii[:,1]-1)), shape=(int(max(ii[:,0]))
,int(max(ii[:,1]))))

readTime=time.perf_counter()

tmp=np.transpose(E)*E
sizeX,sizeY=np.shape(tmp)

print ("Time to Read Data:  " + str(readTime-startTime) + "s")
print ("Computing k-truss")
tmp.setdiag(np.zeros(sizeX),k=0) #A=transpose(E)*E - diag(d)
tmp.eliminate_zeros() #decrease memmory usage by remove zero entries from
 the parse matrix
R= E * tmp

s=lil_matrix(((R==2).astype(float)).sum(axis=1)) #axis=1 convert 2D to 1D
: count of each row of R
xc= (s >=k-2).astype(int)

while xc.sum() != np.unique(sp.sparse.find(E)[0]).shape:
x=sp.sparse.find(xc==0)[0] #find the row < k-2
set_zero_rows(E, x) #set all value of this row equal 0
E=(E>0).astype(int) #update E
tmp=np.transpose(E)*E
(tmp).setdiag(np.zeros(np.shape(tmp)[0]),k=0)
tmp.eliminate_zeros()
R=E*tmp
s=csr_matrix(((R==2).astype(float)).sum(axis=1))
xc= (s >=k-2).astype(int)

ktrussTime=time.perf_counter()
print ("Time to Compute k=truss:  " + str(ktrussTime-startTime) + "s")
return E
```

Time complexity analysis:



The chart above is generated and calculates the median after 10 runs with 18 values of k ranging from 3 to 20. Looking at the chart, we observe that the runtime is not highly dependent on k. This can be demonstrated as follows:

Calculation step: $\mathbf{A} = \mathbf{E}^T\mathbf{E} - diag(\mathbf{E}^T\mathbf{E})$.

Complexity: $O(m.n^2 + m.n + n) = O(m.n^2)$

Calculation step: **R=E.A**

Complexity: $O(m.n^2)$

Calculation step: $s = (\mathbf{R} == 2)\mathbf{1}$

Complexity: $O(m.n^2)$

Calculation step: $x = find(s < k - 2)$

Complexity: $O(m)$

Calculation step: The while loop can execute $O(m)$ times. Within each iteration, $O(m.n^2)$ computations will be performed.

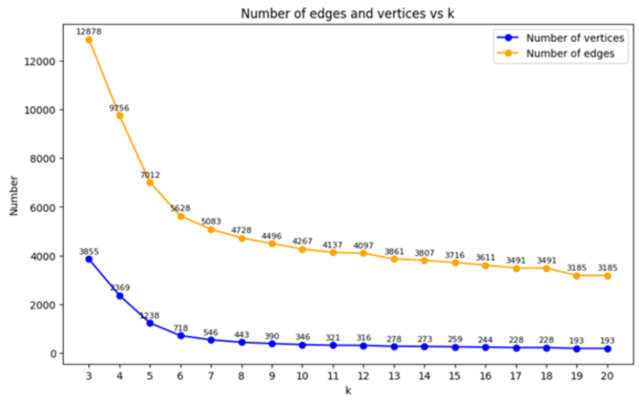
Therefore, the worst-case time complexity of the algorithm is:

$$O(m.n^2) + O(m.n^2) + O(m) + O(m(m.n^2)) = O(m^2.n^2)$$

With m being the number of edges and n being the number of vertices, it can be said that the algorithm has a polynomial time complexity that is not highly dependent on k, as k is only a parameter for comparison in a computation.

Analyze the practical implications for the dataset:

The results for the number of vertices and edges of the k-truss subgraphs obtained with k ranging from 3 to 20:



The Finding Ktruss algorithm holds significant implications for graph analysis. Looking at the chart, the number of vertices and edges in the graph gradually decreases as the k-truss value increases. This aligns with the nature of k-truss, where as the strength of this structure intensifies, the graph becomes sparser with fewer vertices.

For the selected dataset, in the case of each author within a k-truss network, there will be at least $k - 2$ shared research papers with 2 other authors. Authors who participate in at least 18 collaborative research papers amount to only 193 individuals, while those who participate in at least 1 collaborative research paper are much larger in number, totaling 3855 individuals. Due to the large number of authors involved in at least 1 research paper, there will be numerous connections among them, resulting in a higher number of edges. This aligns with reality. Further analysis can be conducted on various aspects such as identifying authors with the highest number of research papers, proposing new research based on the k-truss network with author groups, and more.

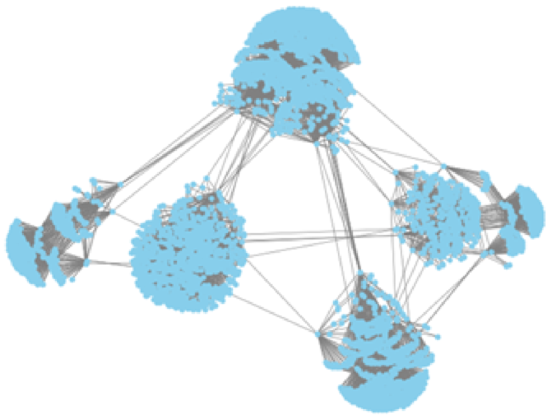


Figure: 20-truss subgraph

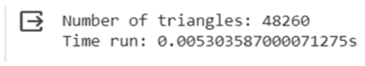
TRIANGLES COUNTING

Code function for the algorithm:

```
 ###################################
 # TRIANGLES COUNTING
 ###################################
 from time import perf_counter
 from glob import glob
 import os,sys
 import numpy as np
 import pandas
 from pandas import read_csv
 from scipy.sparse import csr_matrix
 from scipy.sparse import coo_matrix

 def triangle(adj_mtx_file, inc_mtx_file):

 # figure out the shape of the adjacency matrix
 a = read_csv(adj_mtx_file, sep='\s+', header=None, skiprows=2, nrows=1,
dtype=float).to_numpy()
 M = int(a[0, 0])
 N = int(a[0, 1])

 # read adjacency matrix
 y = read_csv(adj_mtx_file, sep='\s+', header=None, skiprows=3, dtype=
float).to_numpy()

 # convert data to sparse matrix using the coo_matrix function

 A = coo_matrix((y[:, 2], (y[:, 0] - 1, y[:, 1] - 1)), shape=(M, N))

 A = A + A.transpose() #generate symmetric matrix
 adjMtx = A.tocsr() #convert to matrix csr mattrix

 # figure out shape of incidence matrix
```

```
    a = read_csv(inc_mtx_file, sep='\s+', header=None, skiprows=2, nrows=1,
   dtype=float).to_numpy()
    M = int(a[0, 0])
    N = int(a[0, 1])

    # read incidence matrix
    y = read_csv(inc_mtx_file, sep='\s+', header=None, skiprows=3, dtype=
   float).to_numpy()

    # reshape incidence matrix
    B = coo_matrix((y[:, 2], (y[:, 0] - 1, y[:, 1] - 1)), shape=(M, N))
    incMtx = B.tocsr()

    #time start counting
    t0 = perf_counter()
    C = adjMtx @ incMtx
    # count triangles
    num_triangles = (C.data == 2).sum() // 3
    t_triangle_count = perf_counter() - t0

    return (num_triangles, t_triangle_count)
    ####################################################################

```

Executing the algorithm, we obtain the results for the chosen dataset



```
Number of triangles: 48260
Time run: 0.005303587000071275s
```

Number of triangles	48260
Time run	5.3 ms

Correctness: The number of triangles obtained is accurate according to the information provided in the SNAP dataset.

Time complexity analysis:

The matrix multiplication **C=A.E** has a time complexity of $O(m.n^2)$

Counting the elements equal to 2 in matrix **C** requires a time complexity of $O(m.n)$. Therefore, the triangle counting algorithm will have a time complexity of:

$$O(m.n^2) + O(m.n) = O(m.n^2)$$

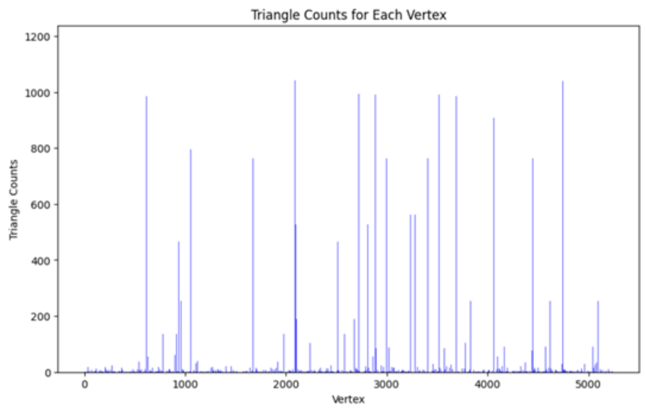
With m being the number of edges and n being the number of vertices.

The algorithm runs in polynomial time.

Analyze the practical implications for the dataset:

The counting of triangles also holds significant importance in graph analysis. A graph with many triangles often exhibits stability and strong connectivity. Analyzing the number of triangles can aid in understanding metrics such as clustering coefficient, transitivity, and other related indices.

From the running results of the dataset, a total of 48260 triangles appear in the graph. In the practical sense of the dataset, this parameter is the number of groups of 3 co-authors participating in a research article which is 48260. This may represent the level of collaboration and linkage between different research groups in a particular field. The ratio of the number of groups with 3 co-authors to the total number of authors is $\frac{48260}{2} \approx 9.2$. It can be understood that on average each author can participate in 9 different groups. The higher this ratio, the greater the level of collaboration and connection between authors in the research community.



An unexplored insight from the algorithm is the concept of overloaded multiplication, aiding in identifying specific triangles within the graph. Counting the number of triangles in each vertex is also crucial. The count of triangles in which each vertex participates is determined by the sum of elements with a value of 2 in each row. In the case of the chosen dataset, the number of triangles at each vertex corresponds to the number of collaborative research papers with two other authors, indicating a single author. If each author participates in multiple research papers, it signals an active research community. Authors engaged in numerous research papers play significant roles in collaborative relationships and may stand out in the research field. In the data set, 2475th authors participate in the most co-research groups, with 1179 groups of 3 co-authors.

Chapter V. Conclusion

In this article, we have solved a small part of the big problem of subgraph isomorphism. Subgraph isomorphism is recognized as an NP-complete problem, meaning that no known polynomial-time algorithm exists to solve it in the general case. The complexity arises from the combinatorial explosion of possible mappings between vertices, requiring a comprehensive exploration of the solution space. Researchers have developed various heuristic and exact algorithms to tackle this complexity, each with its trade-offs in terms of time and space efficiency. We have studied two of them, which are the algorithm for counting triangles and finding the maximal ktruss. Proving the time complexity and determining the running time in practice is the most important thing. We have done and verified this in this article based on a selected dataset. Furthermore, based on two algorithms we have analyzed some implications for this dataset. For the triangle counting algorithm: the number of triangles can be developed for network analysis metrics such as community cohesion, triangle participation ratio,... For the ktruss algorithm, we can verify the characteristics of the ktruss subgraph: as the k value increases, the number of nodes and edges decreases, but the structure and connection between components will become stronger. Hence ktruss can reduce the search space in the given dataset - this is part of the subgraph isomorphism problem.

Acknowledging the limitations of our study is an essential aspect of any research conclusion. We candidly discuss the constraints, challenges, and potential shortcomings encountered during the course of our experiments. This may include data limitations (the given dataset is small with 5242 nodes and 14482 edges) , algorithmic constraints, or any other factors that may have impacted the robustness and generalizability of our findings. Building on the identified limitations, we propose potential avenues for future research. These could involve addressing the shortcomings observed in our current study, exploring new applications of the proposed methods, or delving into related areas that warrant further investigation. Because knowledge is limited and the report is only written in the form of a major assignment for the subject, this needs to be accumulated from the specialized knowledge that we will learn in the future.

References

[1]Kenneth H. Rosen, *“Discrete Mathematics Applications and Its Applications”*, Eighth Edition

[2]Byjus, “Adjacency Matrix”, Adjacency Matrix - Definition, Properties, Theorems, Graphs and Example (byjus.com)

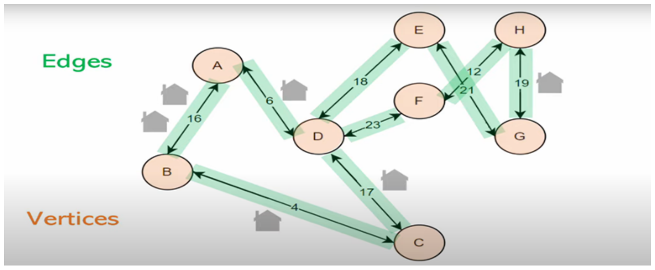
[3] Katana Graph, “K-Core and K-Truss Algorithms

[4] MIT Lincoln Laboratory, Lexington, MA, “Static Graph Challenge: Subgraph Isomorphism”

[5] Jonathan Cohen. “Trusses: Cohesive subgraphs for social network analysis”. National Security Agency Technical Report.

Describe the Chinese postman problem and explain how to solve this problem.

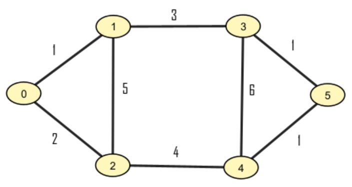
Chapter I. Introduction



The Chinese Postman Problem (CPP) is a fascinating conundrum in graph theory that finds its roots in the world of postal services and network optimization. First introduced by Kwan Mei-Ko in 1962, this problem has since captured the attention of mathematicians, computer scientists, and logistics experts alike. The essence of the CPP lies in its practical applications, specifically in optimizing the routes taken by postmen to deliver mail efficiently, minimizing the overall cost and distance traveled. At its core, the Chinese Postman Problem deals with finding the most cost-effective way to traverse each edge of a graph at least once while returning to the starting point. The graph may represent a network of streets, roads, or any interconnected system where each edge represents a segment that needs to be covered. The challenge is to find a closed circuit that includes all edges with the minimum possible total cost.

The real-world significance of this problem becomes apparent when considering scenarios where postal services or maintenance crews need to cover all streets in a city or a specific region. In these situations, an optimal route not only saves time and resources but also ensures a more sustainable and environmentally friendly approach.

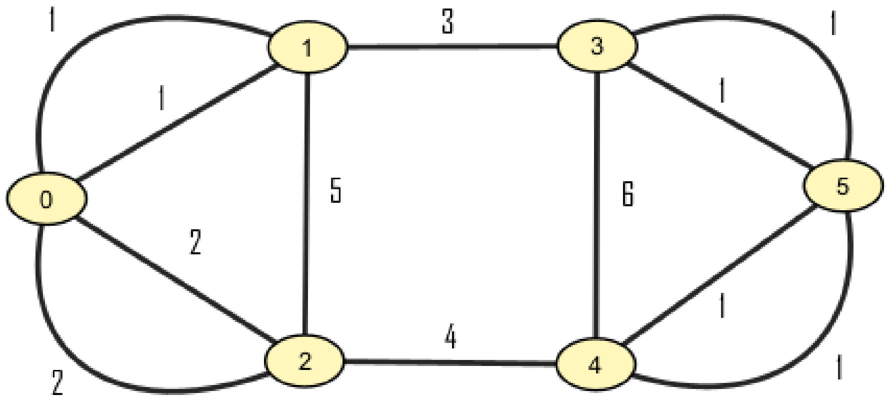
Chaptere II. Solving the Chinese Postman Problem



To solve this problem we need to know about the Euler cycle and the Euler path. **An Euler cycle** is a graph in which no vertex has an odd degree. The cycle in allows the path at every vertex to move to all edges at least once and end at the starting vertex. If there is an Euler cycle, there will be an Euler path. **An Euler path** is a path that can start at a vertex with odd degree and end at another vertex with odd degree.

If the problem is a graph with an Euler cycle, the result for the problem is the sum of all weighted edges on the graph. Otherwise, the problem will be solved according to the following steps:

1. Create as many pairs as possible whose components are vertex pairs. Like this: [(1,2), (3,4)], [(1,3), (2,4)], [(1,4), (2,3)].
2. Find the pair with the smallest sum between the components.
   (1,2) = (1,0) + (0,2) = 1 + 2 = 3
   (3,4) = (3,5) + (4,5) = 1 + 1 = 2
   Total = 3 + 2 = 5.
3. Create a new graph with edges through the selected vertices to form a graph with an Euler cycle.



4. Print out the Euler cycle with 2 vertices redrawn with the starting and ending points being those 2 vertices.

Totally, you can solve this with algorithm:

Step 1: If graph is Eulerian, return sum of all edge weights. Else do following steps.

Step 2: We find all the vertices with odd degree

Step 3: List all possible pairings of odd vertices. For n odd vertices total number of pairings possible are, (n-1) * (n-3) * (n -5) ... * 1

Step 4: For each set of pairings, find the shortest path connecting them.

Step 5: Find the pairing with minimum shortest path connecting pairs.

Step 6: Modify the graph by adding all the edges that have been found in step 5.

Step 7: Weight of Chinese Postman Tour is sum of all edges in the modified graph.

Step 8: Print Euler Circuit of the modified graph. This Euler Circuit is Chinese Postman Tour.

Chapter III. Conclusion

The Chinese Postman Problem is a classic optimization problem in graph theory. It involves finding the most efficient way for a postman to deliver mail to every house on their route while minimizing the total distance traveled. This problem has applications in various fields, and several approaches can be used to solve it. Here are some applications:

1. **Mail Delivery:** In the context of the original problem, the Chinese Postman Problem can be applied to optimize the route of a mail carrier to deliver mail to every address on their route while minimizing the distance traveled.

2. **Network Maintenance:** The problem can be used to optimize the routes of maintenance crews responsible for checking and maintaining various components in a network, such as power lines, pipelines, or communication networks.

3. **DNA Sequencing:** In bioinformatics, the Chinese Postman Problem can be used to optimize the sequencing of DNA fragments, where the goal is to find the shortest possible sequence that contains all given fragments.

In summary, solving the Chinese Postman Problem offers practical benefits in terms of optimizing routes, reducing operational costs, improving service quality, and addressing resource utilization challenges in various industries, particularly those involving delivery and transportation.

References

[1]GeekforGeek "*Chinese Postman or Route Inspection*" Chinese Postman or Route Inspection | Set 1 (introduction) - GeeksforGeeks.

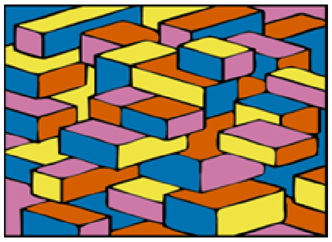
[2]Grace Wang "The Chinese Postman Problem (Introduction to Graph Theory)", The Chinese Postman Problem (Introduction to Graph Theory) - YouTube.

[3] Wikipedia "Chinese postman problem" Chinese postman problem - Wikipedia.

Discuss the history of The Four Color Theorem

Chapter I. Introduction

The Four Color Theorem, one of the most intriguing enigmas in mathematics, has captivated the curiosity and interest of leading researchers from the 19th century to the present day. In mathematics, the four color theorem, or the four color map theorem, states that no more than four colors are required to color the regions of any map so that no two adjacent regions have the same color. Adjacent means that two regions share a common boundary curve segment, not merely a corner where three or more regions meet. In this essay, we delve into the history of this theorem, exploring its early developments, the challenges faced, and the ultimately triumphant journey toward a transparent proof.[1]


Example of a four-colored map 1

Chapter II. Mathematical foundations and history

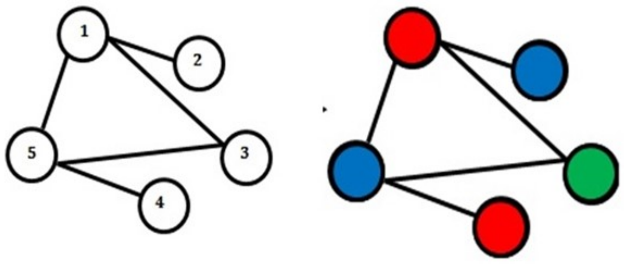
A.Fundamental Concepts in Graph Theory

Graph Coloring Problem Graph coloring, specifically vertex coloring, involves assigning colors to each vertex of a graph in such a way that no two adjacent vertices share the same color, and the minimum number of colors are used. The minimum number of colors needed to color a graph is known as its chromatic number. Algorithms for graph coloring are discussed in discrete mathematics or data structures & algorithms literature. The algorithmic steps can be presented in an understandable manner as follows:

Input: Graph G = (V, E).

Output: Graph G = (V, E) with vertices colored.

Algorithm Steps:



Example: Consider the graph in the illustration. Using the graph coloring algorithm outlined above, color the vertices of the graph.

Step 1: The graph has 5 vertices labeled 1, 2, 3, 4, 5 with corresponding degrees 3, 1, 2, 1, 3. Hence, V' initially has the order [1, 5, 3, 2, 4]. Set i = 1.

Step 2: Color vertex 1 with color 1 (red). Iterate through the remaining vertices in V':

- Vertex 5 is adjacent to vertex 1 (colored with 1 - red), so it remains uncolored. Similarly, vertices 3 and 2 are adjacent to vertex 1, so they are also uncolored.
- Vertex 4 is not adjacent to vertex 1, so color vertex 4 with color 1 (red). Vertex 4 is now colored with 1 - red.

Step 3: Check that there are still uncolored vertices in V'; proceed to step 4.

Step 4: Remove colored vertices 1 and 4 from V', reorder V' in non-increasing degree order, yielding V' = [5, 3, 2]. Set i = 2 and repeat step 2.

Step 2(1): Color vertex 5 with color 2 (blue). Iterate through remaining vertices in V':

- Vertex 3 is adjacent to vertex 5 (colored with 2 - blue), so it remains uncolored.
- Vertex 2 is not adjacent to vertex 5, so color vertex 2 with color 2 (blue). Vertex 2 is now colored with 2 - blue.

Step 3(1): Check that vertex 3 is still uncolored; proceed to step 4(1).

Step 4(1): Remove colored vertices 5 and 2 from V', yielding V' = [3]. Set i = 3 and repeat step 2.

Step 2(2): Color vertex 3 with color 3 (green).

Step 3(2): Check that all vertices in V are colored; the algorithm stops.

Conclusion: Vertices 1 and 4 are colored with 1 - red, vertices 5 and 2 are colored with 2 - blue, and vertex 3 is colored with 3 - green. The minimum number of colors required is i = 3.

The graph coloring problem finds practical applications in solving problems like map coloring (countries), scheduling exams, and room assignments.[2]

B.Early History of the Problem

As far as is known, the conjecture was first proposed on October 23, 1852, when Francis Guthrie, while trying to color the map of counties of England, noticed that only four different colors were needed. At the time, his brother Frederick Guthrie had become a student of De Morgan. Francis Guthrie showed his brother some results he had been trying to prove about the colouring of maps and asked Frederick to ask De Morgan about them.[3]

C.Initial proof attempts

The quest to solve Guthrie's problem attracted the attention of several mathematicians, each contributing to the exploration of the Four Color Conjecture. Charles Peirce in the United States embarked on an attempt to prove the Conjecture in the 1860s, maintaining a lifelong interest in the problem. In parallel, Augustus De Morgan continuously sought solutions and encouraged fellow mathematicians to tackle the challenge.

In 1878, Arthur Cayley, having learned of the problem from De Morgan, inquired with the London Mathematical Society about the status of the Four Colour Conjecture. Subsequently, on June 13, 1878, Cayley presented a paper titled "On the colouring of maps"to the Royal Geographical Society, which was later published in 1879. This paper detailed the complexities involved in attempting to prove the Conjecture.

A significant development occurred on July 17, 1879, when Alfred Bray Kempe declared in the journal Nature that he had a proof for the Four Colour Conjecture. Kempe's approach involved the use of an argument known as the method of Kempe chains. The essence of this method lies in the examination of a map where every region is colored with one of four colors, except for a single region, X. Kempe reasoned that if X is not surrounded by regions of all four colors, then there is a color left for X. However, if X is surrounded by regions A, B, C, D in order, colored red, yellow, green, and blue, there are two cases to consider:

- (I) There is no chain of adjacent regions from A to C alternately colored red and green.

- (II) There is a chain of adjacent regions from A to C alternately colored red and green.

If (I) holds there is no problem. Change A to green, and then interchange the colour of the red/green regions in the chain joining A. Since C is not in the chain it remains green and there is now no red region adjacent to X. Colour X red. If (II) holds then there can be no chain of yellow/blue adjacent regions from B to D. [It could not cross the chain of red/green regions.] Hence property (i) holds for B and D and we change colours as above.

Kempe's announcement brought excitement and optimism, but later scrutiny revealed flaws in his proof. This episode marked a crucial stage in the ongoing quest to conclusively address the Four Color Conjecture.

In 1890, in addition to exposing the flaw in Kempe's proof, Heawood proved the five color theorem and generalized the four color conjecture to surfaces of arbitrary genus.

Tait, in 1880, showed that the four color theorem is equivalent to the statement that a certain type of graph (called a snark in modern terminology) must be non-planar.

In 1943, Hugo Hadwiger formulated the Hadwiger conjecture, a far-reaching generalization of the four-color problem that still remains unsolved.[4]

D.Final Proof

During the 1960s and 1970s, German mathematician Heinrich Heesch developed methods of using computers to search for a proof. Notably he was the first to use discharging for proving the theorem, which turned out to be important in the unavoidability portion of the subsequent Appel–Haken proof. He also expanded on the concept of reducibility and, along with Ken Durre, developed a computer test for it. Unfortunately, at this critical juncture, he was unable to procure the necessary supercomputer time to continue his work. Others took up his methods, including his computer-assisted approach. While other teams of mathematicians were racing to complete proofs, Kenneth Appel and Wolfgang Haken at the University of Illinois announced, on June 21, 1976, that they had proved the theorem. They were assisted in some algorithmic work by John A. Koch.

If the four-color conjecture were false, there would be at least one map with the smallest possible number of regions that requires five colors. The proof showed that such a minimal counterexample cannot exist, through the use of two technical concepts:

- An unavoidable set is a set of configurations such that every map that satisfies some necessary conditions for being a minimal non-4-colorable triangulation (such as having minimum degree 5) must have at least one configuration from this set.

- A reducible configuration is an arrangement of countries that cannot occur in a minimal counterexample. If a map contains a reducible configuration, the map can be reduced to a smaller map. This smaller map has the condition that if it can be colored with four colors, this also applies to the original map. This implies that if the original map cannot be colored with four colors the smaller map cannot either and so the original map is not minimal.

Using mathematical rules and procedures based on properties of reducible configurations, Appel and Haken found an unavoidable set of reducible configurations, thus proving that a minimal counterexample to the four-color conjecture could not exist. Their proof reduced the infinitude of possible maps to 1,834 reducible configurations (later reduced to 1,482) which had to be checked one by one by computer and took over a thousand hours. This reducibility part of the work was independently double checked with different programs and computers. However, the unavoidability part of the proof was verified in over 400 pages of microfiche, which had to be checked by hand with the assistance of Haken's daughter Dorothea Blostein.

Appel and Haken's announcement was widely reported by the news media around the world, and the math department at the University of Illinois used a postmark stating "Four colors suffice."At the same time the unusual nature of the proof—it was the first major theorem to be proved with extensive computer assistance—and the complexity of the human-verifiable portion aroused considerable controversy.

In the early 1980s, rumors spread of a flaw in the Appel–Haken proof. Ulrich Schmidt at RWTH Aachen had examined Appel and Haken's proof for his master's thesis that was published in 1981. He had checked about 40% of the unavoidability portion and found a significant error in the discharging procedure In 1986, Appel and Haken were asked by the editor of Mathematical Intelligencer to write an article addressing the rumors of flaws in their proof. They replied that the rumors were due to a "misinterpretation of [Schmidt's] results"and obliged with a detailed article. Their magnum opus, Every Planar Map is Four-Colorable, a book claiming a complete and detailed proof (with a microfiche supplement of over 400 pages), appeared in 1989; it explained and corrected the error discovered by Schmidt as well as several further errors found by others.[5]

Chapter III. Conclusion

In conclusion, the history of the Four Color Theorem is a fascinating journey that spans centuries and involves the contributions of numerous mathematicians. From the initial formulation of the conjecture by Francis Guthrie in 1852 to the groundbreaking proof by Kenneth Appel and Wolfgang Haken in 1976, the theorem has captured the imagination of the mathematical community and the general public alike.

The early attempts to prove the conjecture, including those by prominent mathematicians like Alfred Bray Kempe and Percy Heawood, set the stage for further exploration and inquiry. The formulation of the problem in terms of graph theory, the introduction of new concepts such as chromatic numbers and planar graphs, and the application of advanced mathematical methods all played crucial roles in advancing our understanding of the problem. The collaborative efforts of mathematicians across different eras and continents reflect the global nature of mathematical inquiry. The integration of computer-assisted approaches in the 20th century marked a significant shift in the methodology used to tackle complex mathematical problems, with Appel and Haken's proof being a landmark example of the synergy between human intuition and computational power.

Despite the theorem's proof and acceptance, the controversy surrounding its extensive computer-assisted nature has left a lasting impression on the field of mathematics. The Four Color Theorem continues to be a symbol of both the triumphs and challenges inherent in mathematical research. As we reflect on its history, we recognize the theorem's profound impact on the development of graph theory and its applications in various fields, underscoring the interconnectedness of mathematical ideas and their enduring relevance.

References

[1] J J O'Connor and E F Robertson." The four colour theorem" September 1996.

[2] Nguyen Tuan Nghia. "Graph coloring" 14:33 29/06/2018.

[3] "The Four-Color Theorem: A History" Revised: 25/4/2023.

[4] WonfarmMathWorld "Four-Color Theorem" Wed Nov 29 2023.

[5] Wikifedia "Four color theorem" https://en.wikipedia.org/wiki/Four_color_theorem#Proof_by_computer 13 November 2023, at 21:01 (UTC).