The Study of Vertex Coloring Algorithms Using Heuristic Approaches

By

Pratik Lodha
201611016

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
In
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

May, 2018

Declaration

I hereby declare that

i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,

ii) due acknowledgment has been made in the text to all the reference material used.

_____

Pratik Lodha

Certificate

This is to certify that the thesis work entitled The Study of Vertex Coloring Problem Using Heuristic Approaches has been carried out by Pratik Lodha for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my supervision.

_____

Rahul Muthu
Thesis Supervisor

Acknowledgments

Undertaking this Masters at DAIICT has been a truly a life-changing experience for me. Last one-year was the journey of lots of up's and down's. And it would not be possible to reach this final stage without the guidance and support of my supervisor **Prof. Rahul Muthu**. He has taught me many important lessons right from the beginning during his course on 'Graph Theory' till the completion of my thesis. From the beginning of our thesis time-line he gave me complete freedom from choosing my own problem statement, then walked me through form where to begin, how to approach the problem, then solving doubts at each and every stage and till this final thesis submission he is completely by my side.

I would also like to thank **Prof. Saurabh Tiwari** for guiding me throughout all the stages. My problem statement was quiet difficult but his constant motivation and kind words after each stage presentation boosted my morale to work without losing my confidence. He is truly dedicated towards his work and found very responsible in guiding me time to time. I saw passion for his work when even though he was out of the country he attended my Stage-3 presentation through Skype. I truly admire his sincerity.

I would like to say a very big thank you to my Dad, Mom and Mayank for their love and emotional support. I would also like to thank my friends Lokesh Kumar, Heli Sheth and senior Dhaval Patel, who were always there in my moments of doubts and confusion. At last I would also like to thank Resource Center staff for maintaining decorum, silence and discipline on all days.

Contents

Abstract **v**

List of Figures **1**

1 Introduction **2**
1.1 Definition 2

2 Previous Algorithms **4**
2.1 Greedy Vertex Coloring. 4
2.2 Welsh-Powell Greedy Algorithm. 4
2.3 Degree Sequencing. 5

3 Proposed Algorithms **6**
3.1 Eccentricity Based Vertex Coloring. 6
3.2 DFS Based Coloring. 7
3.3 Maximum Degree Based Coloring 8

4 Proposed Algorithms Analysis **9**
4.1 Graph Classes. 9
4.2 Algorithms Comparison. 11

5 Conclusion **22**

6 Future Work **23**

References **24**

Appendix **25**

Abstract

Graph vertex coloring is one of the most studied NP-complete optimization problem (READ, 1972) [2]. The problem is that; given a graph G, determine the number of colors required to color G, so that no two adjacent vertices share the same color. And the minimum number of colors required to color graph is known as **Chromatic Number** and is denoted by $\chi(G)$. By using existing properties of eccentricity, BFS, DFS (West, 2000) [3] and graph components we have proposed three new heuristic algorithms to obtain approximated chromatic number of a given graph G. And these approaches are as follows:

1. Eccentricity based coloring
2. DFS based coloring, and
3. Maximum degree based coloring.

List of Figures

4.1 Harary Line Graph. 11
4.2 Kneser Line Graph.. 13
4.3 Hyper-cube Line Graph. 15
4.4 Star Line Graph. 16
4.5 Path Line Graph. 17
4.6 Wheel Line Graph. 18
4.7 Complete k-partite Line Graph. 19
4.8 Cycle Line Graph. 21
7.1 Eccentricity Vertex Coloring. 25
7.2 DFS Vertex Coloring. 26
7.3 Maximum Degree based Coloring. 28

CHAPTER 1

Introduction

Vertex coloring is the assignment of colors to each vertex of the graph such that no two adjacent vertices share the same colors. This vertex coloring problem seeks to optimize the number of required colors. Such coloring is known as a minimum vertex coloring (A.Lim, Y. Zhu, & Q. Lou, 2005) [4], and the minimum number of colors required to color a graph G is called as the Chromatic Number, denoted as $\chi(G)$.

A minimum vertex coloring (Weisstein) [7] can be computed using Brelaz's heuristic algorithm. This algorithm can find good, but not necessarily minimal vertex coloring of a graph. It is known and implemented as Brelaz coloring in the Wolfram Language Package Combinatorical.

The k-coloring of a graph with k or fewer numbers of colors is known as vertex coloring with k-colors. And such graph having k-coloring is said to be a k-colorable, while the graph which is k-colorable is also known as k-chromatic. Below is the definition section which is very essential ingredients of our proposed algorithms.

1.1 Definition

Definition 1. ***Graph*** *G is a set (V, E), where V is the set of vertices and E represents set of edges connecting a pair of vertices.*

Definition 2. *The* ***eccentricity*** *is a node centrality index. To calculate eccentricity of a vertex first computes shortest distance from that vertex to all the other vertices in the graph. Then from the list of distances choose the highest integer value and this value represents eccentricity of that vertex.*

Definition 3. ***Depth-first search (DFS)*** *is a graph traversing algorithm. In this we start our traversal at an arbitrary graph vertex and explore the graph as far as possible before backtracking to the explored vertex and again traversing as far as possible till all the vertices of the graph are explored/ traversed.*

Definition 4. ***Breadth-first search (BFS)*** *is a graph traversing algorithm. In this algorithm we start at an arbitrary vertex of a graph and explore the neighbor nodes first, before moving to explore next level neighbor. BFS is also known as level wise neighbor exploration algorithm.*

Definition 5. *An* ***independent set*** (Graph Coloring - Wikipedia) [6] *or stable set is a set of vertices in a graph, no two vertices from the same set are adjacent to each other, i.e. for any given set if the intra-set edge is absent then that set is called as independent set. In our graph coloring optimization problem our sole aim is to find minimum number of these independent set.*

CHAPTER 2
Previous Algorithms

In this chapter we present some of the known algorithms that I studied as a part of literature survey (Brélaz, 1979) [1]. To get the feel and understanding of how graph chromatic algorithms work I implemented them in Java programming language.

Algorithm 1 Greedy Vertex Coloring

Input: Graph

Output: Chromatic Number

1. Choose a vertex and color it with first color.
2. Do following for remaining v – 1 vertices:
   Choose a vertex v from V and color it with lowest available color such that it is not yet used on the adjacent vertices of v. If color that is previously used appears on vertices adjacent to v then assign a new color to it.

Algorithm 2 Welsh-Powell Vertex Coloring

Input: Graph

Output: Chromatic Number

1. Sort the vertices in G according to decreasing value of their degree. Place this sorted list in an array V.
2. Consider a list C and place all the available colors in C.
3. In this strategy the first non-colored vertex form V is picked and colored with the first available color that is not previously used.
4. Perform step-3 till there are no vertices in V left uncolored. Else FINISH.

Algorithm 3 Using Degree Sequencing

It provides a better strategy for coloring a graph. It chooses a vertex for coloring based on a selection criterion. This strategy is better than 'Greedy coloring', which simply picks a vertex from an arbitrary order. Here are some strategies for selecting the next vertex to be colored (Heuristicswiki - home) [5]:

Input: Graph

Output: Chromatic Number

A. Largest Degree Ordering (LDO)
In this strategy we list the vertex in descending order based on its degree and color it applying greedy coloring on the sequence. Rather than picking some arbitrary vertex it provides a sequence to pick-up a vertex to be colored.

B. Shortest Degree Ordering (SDO)
In this strategy we list the vertex in ascending order based on its degree and color it applying greedy coloring on the sequence. Rather than picking some arbitrary vertex it provides a sequence to pick-up a vertex to be colored. This strategy performs poor as compared to LDO.

CHAPTER 3

Proposed Algorithms

In this chapter we will see some of the heuristic vertex coloring algorithms. By definition, Heuristics provide approximated solution to the range of NP problems. In this section our main motto is to come up with the solutions in the form of algorithms such that it yields value as close as to our ideal solution (i.e. Chromatic Number).

Algorithm 1 Eccentricity Based Vertex Coloring

Input: Graph

Output: Approximate Chromatic Number

1. Consider the given graph G.
2. Compute the eccentricity of all vertices.
3. Consider the vertex induced sub graph H of all highest eccentricity vertices (i.e. diameter).
4. Create a set (i.e. Independent set) containing all disconnected vertices.
5. Remove S_1 vertices from the G. Repeat step 2 to step 4, till there are no vertices in the graph.
6. Perform exhaustive union of all the obtained sets.

Algorithm 2 DFS Based Vertex Coloring

Input: Graph

Output: Chromatic Number

1. Find the complement of a given graph G.
2. Run Depth-First Search (DFS) on this complement graph.
3. Since we have obtained DFS on the complement graph note that each edge represents a set of non-adjacent vertices from the original graph G.
4. Consider each edge "e_{ij}" of DFS tree where i, j represents two endpoints of the corresponding edge in the form of a set S_k = (i, j), where k = Number of edges in DFS tree.
   NOTE: The sets obtained in STEP 4 represent independent sets and we can reduce the number of independent sets by applying step 5.
5. List down all such sets and perform exhaustive union operation on these sets till no more sets are union-able. Two sets are union able if and only if no edge exists from one set to other set.
6. The final number of independent set represents approximate or heuristic value required to color that G.

Algorithm 3 Maximum Degree Based Coloring

In this section, we will see a vertex coloring heuristics algorithm by splitting the given graph into a number of components using the maximum degree vertex.

Input: Graph

Output: Chromatic Number

1. Given a graph G, locate the maximum degree vertex in G. In case if there is more than one vertex with the maximum degree then choose a vertex in their ascending order.
2. Remove this maximum degree vertex along with all its adjacent vertices. Now there is at least one component in the resultant graph. And each component has at least one vertex.
   (Note: A single component is nothing but a group of connected vertices or in other words, each component represents disconnected subgraph of a given graph)
3. Now put maximum degree vertex obtained in STEP 2 along with one vertex from each component in a set S_i. Note that set Si represents an independent set.
4. Remove all the vertices of set S_i from G. We will get a new graph G_j.
5. Apply STEP 1 to STEP 4 on graph G_j. And obtain next corresponding independent set. Perform STEP 1 to STEP 4 till there are no vertices in G.
6. Perform exhaustive union operation on all the obtained sets. Thus the final number of independent set represents approximated chromatic number of graph G.

CHAPTER 4

Proposed Algorithms Analysis

This chapter consists of two sections. In section 4.1 we will see some of the common classes of graphs along with their definition. In section 4.2 we will test and analyze our proposed algorithms against all the classes of graph described in section 4.1.

4.1 Graph Classes

A. Harary Graph (Extended) [8]

The extended Harary graph $H_{k,n}$ is a k-connected graph with n vertices having the smallest possible number of edges. Given k < n, place n vertices around a circle and if:

- k = even: Make each vertex adjacent to $\frac{k}{2}$ vertices in each direction of circle.
- k = odd & n = even: Make each vertex adjacent to nearest $k - \left(\frac{1}{2}\right)$ vertices in clockwise and anti-clockwise direction and also to diametrically opposite vertex.
- k & n = even: Index the vertices by integer modulo n. Construct Hk,n from Hk-1,n by adding the edges as,

$$i = i + \frac{n-1}{2}, \text{ for } 0 \leq i \leq \frac{n-1}{2}$$

B. Kneser Graph

In graph theory, the Kneser graph $G_{n,k}$ is the graph whose vertices correspond to the k-element subsets of a set of n elements, and where two vertices are adjacent if and only if the two corresponding sets are disjoint.

C. Hyper-cube Graph

The n-hypercube graph commonly denoted as Q_n, is a graph whose vertices are 2^n symbols, where each symbol can either be 0 or 1 and two vertices are adjacent iff the symbols differ in exactly one coordinate.

D. Star Graph

A star graph is a complete bipartite graph $K_{1,k}$, in which there is only one vertex in one of the independent set which is connected to all the vertices of other independent set.

E. Path Graph

A path graph P_n is a tree in which two nodes known as end vertices have degree one and all other vertices have degree two. A path graph therefore can be formed such that all its vertices and edges are alternately placed in a sequence to form a chain.

F. Wheel Graph

A graph which is formed by connecting a single vertex to all vertices of a cycle is a wheel graph.

G. Complete k-partite Graph

A graph in which all the vertices are grouped in k independent sets. Moreover, a vertex from one group is adjacent to all the vertices of other groups and the vertexes from a group are non-adjacent to each other in the same group i.e. intra-group edge is absent. Such a graph is known as complete k-partite graph.

H. Cycle Graph

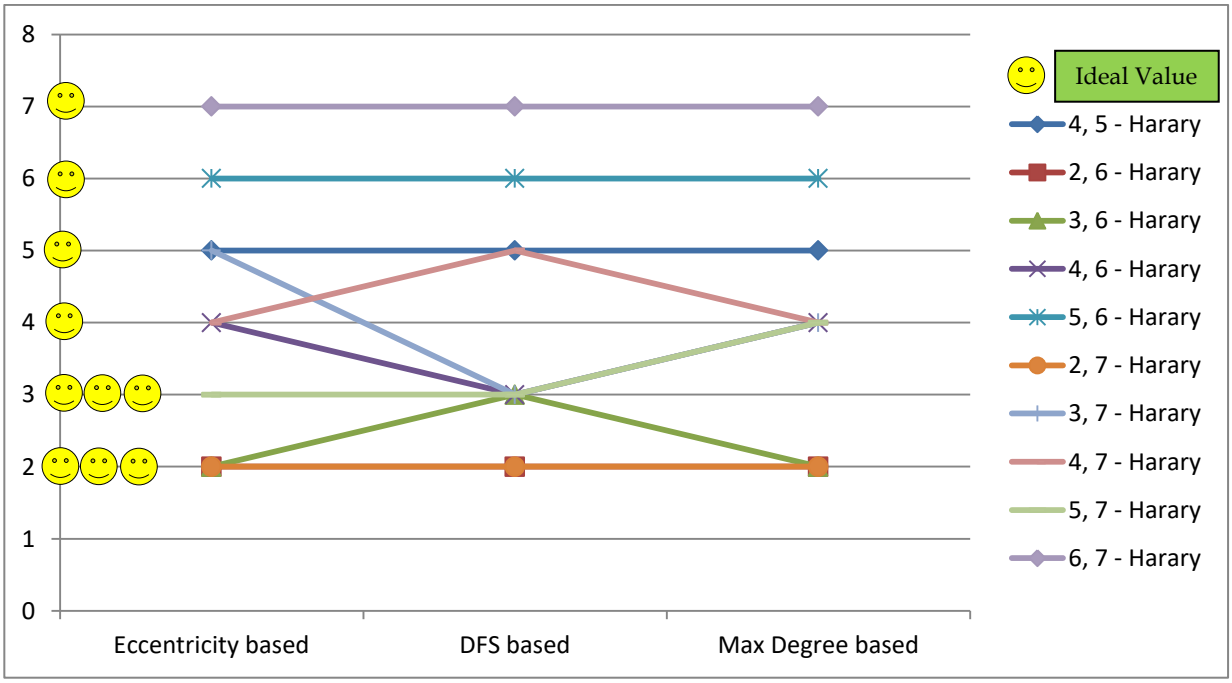
A graph in which all the vertices are connected in a closed chain such that the last vertex in chain is connected to the first vertex and degree of all the vertices is exactly 2.

4.1 Algorithms Comparison

A. Harary Graph ($H_{k,n}$)

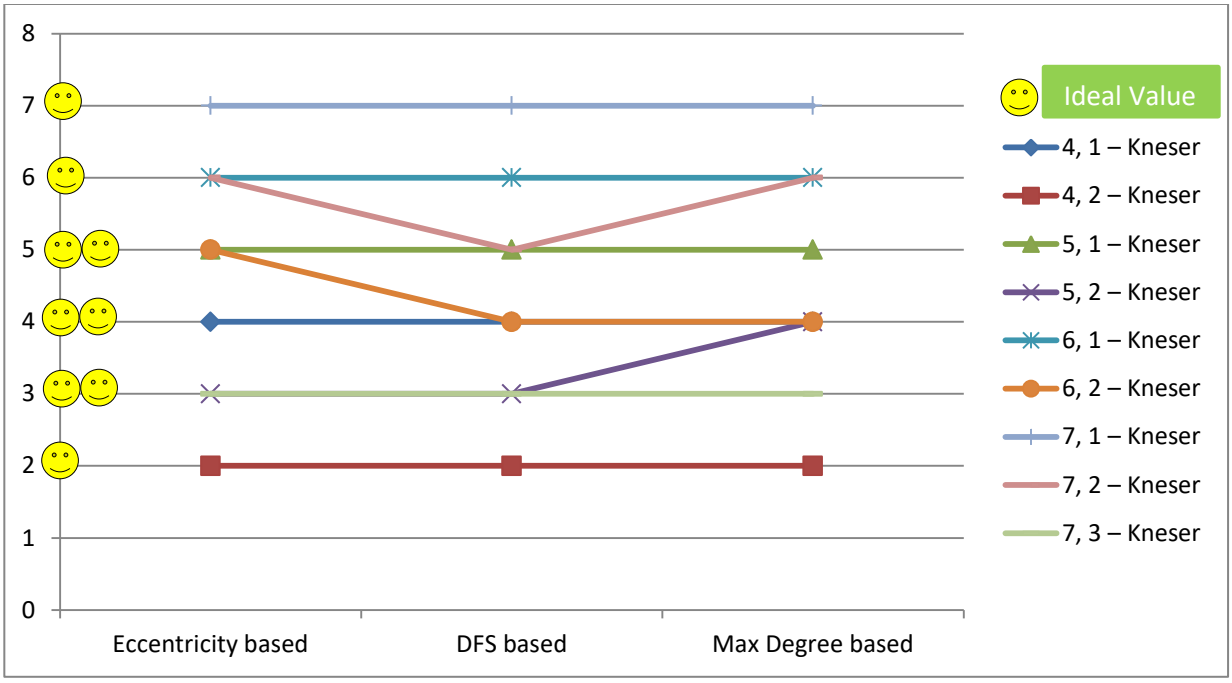
4, 5 – Harary Graph	Our Algorithm Value	Ideal / Chromatic number
Eccentricity Based	5	5
DFS Based	5	
Maximum Degree Based	5	
2, 6 – Harary Graph		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
3, 6 – Harary Graph		2
Eccentricity Based	2	
DFS Based	3	
Maximum Degree Based	2	
4, 6 – Harary Graph		3
Eccentricity Based	4	
DFS Based	3	
Maximum Degree Based	4	
5, 6 – Harary Graph		6
Eccentricity Based	6	
DFS Based	6	
Maximum Degree Based	6	
2, 7 – Harary Graph		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
3, 7 – Harary Graph		3
Eccentricity Based	4	
DFS Based	3	
Maximum Degree Based	5	
4, 7 – Harary Graph		4
Eccentricity Based	4	
DFS Based	5	
Maximum Degree Based	4	
5, 7 – Harary Graph		3
Eccentricity Based	4	
DFS Based	3	

Maximum Degree Based	3	
6, 7 – Harary Graph		7
Eccentricity Based	7	
DFS Based	7	
Maximum Degree Based	7	



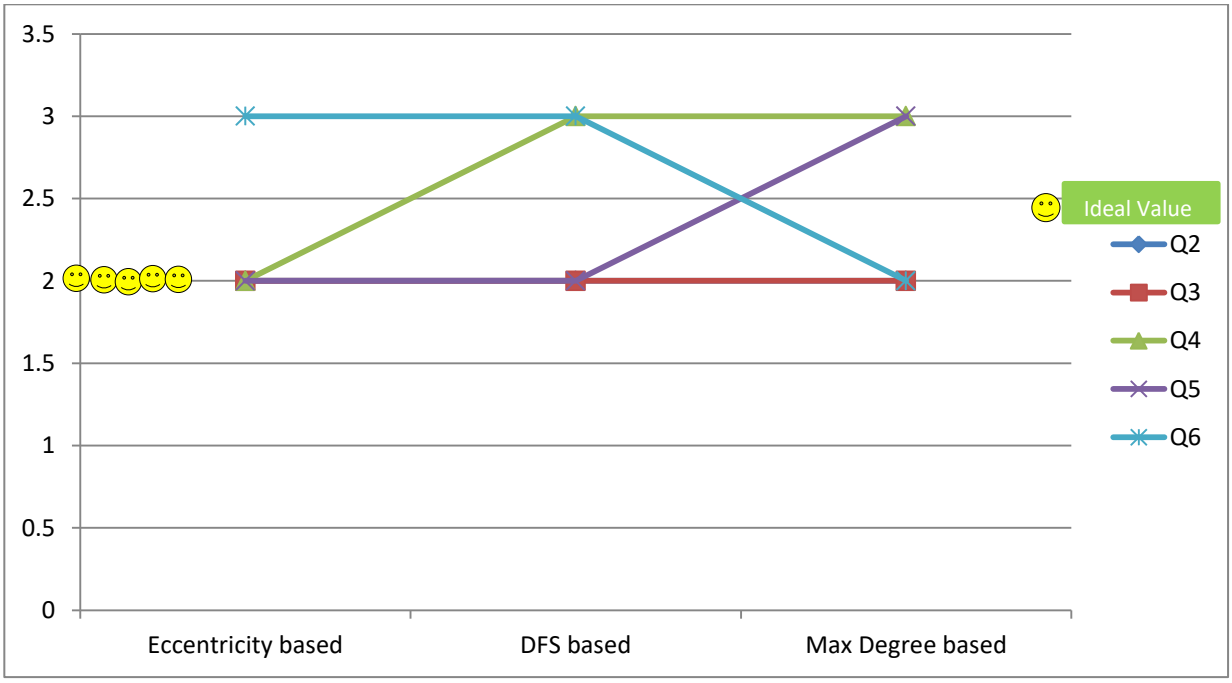
B. Kneser Graph ($\chi(G)$: n – 2k + 2, if n>2k else 1)

4, 1 – Kneser Graph	Our Algorithm Value	Ideal / Chromatic number
Eccentricity Based	4	4
DFS Based	4	
Maximum Degree Based	4	
4, 2 – Kneser Graph		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
5, 1 – Kneser Graph		5
Eccentricity Based	5	
DFS Based	5	
Maximum Degree Based	5	
5, 2 – Kneser Graph		3
Eccentricity Based	4	
DFS Based	3	
Maximum Degree Based	3	
6, 1 – Kneser Graph		6
Eccentricity Based	6	
DFS Based	6	
Maximum Degree Based	6	
6, 2 – Kneser Graph		4
Eccentricity Based	4	
DFS Based	4	
Maximum Degree Based	5	
7, 1 – Kneser Graph		7
Eccentricity Based	7	
DFS Based	7	
Maximum Degree Based	7	
7, 2 – Kneser Graph		5
Eccentricity Based	6	
DFS Based	5	
Maximum Degree Based	6	
7, 3 – Kneser Graph		3
Eccentricity Based	3	
DFS Based	3	
Maximum Degree Based	3	


8
7
6
5
4
3
2
1
0
Eccentricity based
DFS based
Max Degree based
Ideal Value
4, 1 – Kneser
4, 2 – Kneser
5, 1 – Kneser
5, 2 – Kneser
6, 1 – Kneser
6, 2 – Kneser
7, 1 – Kneser
7, 2 – Kneser
7, 3 – Kneser

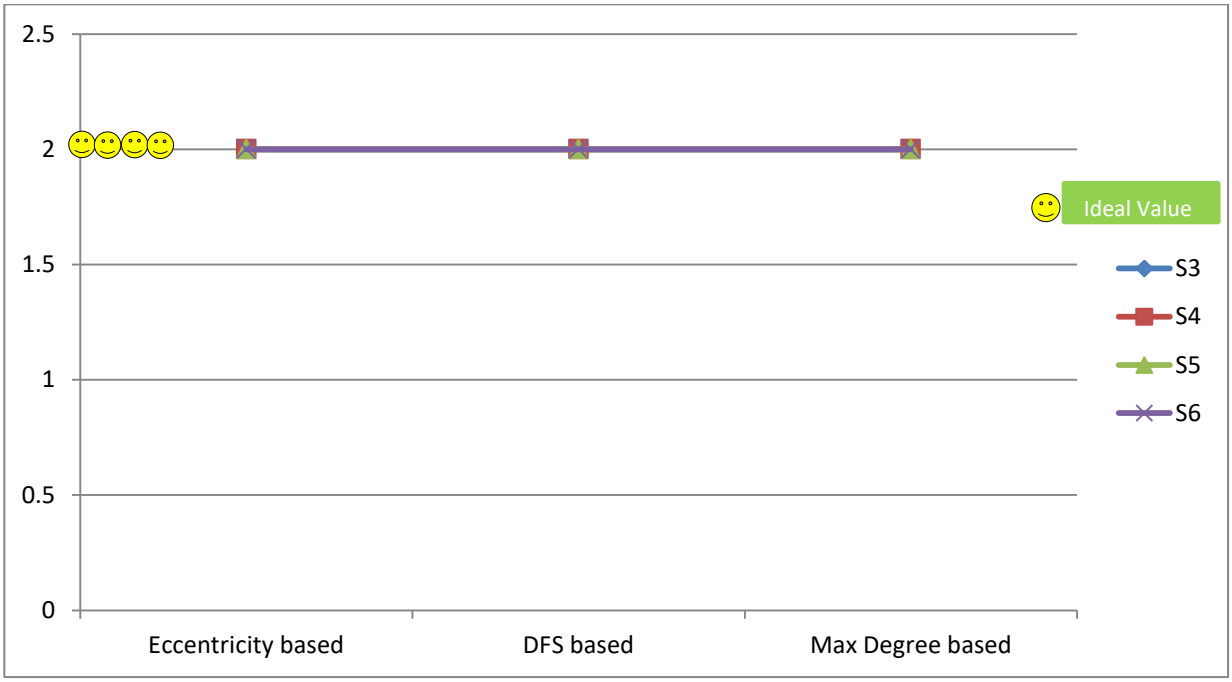
C. Hypercube Graph (Q_n)

Q_2	Our Algorithm Value	Ideal / Chromatic number
Eccentricity Based	2	2
DFS Based	2	
Maximum Degree Based	2	
Q_3		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
Q_4		2
Eccentricity Based	3	
DFS Based	3	
Maximum Degree Based	2	
Q_5		2
Eccentricity Based	3	
DFS Based	2	
Maximum Degree Based	2	
Q_6		2
Eccentricity Based	2	
DFS Based	3	
Maximum Degree Based	3	



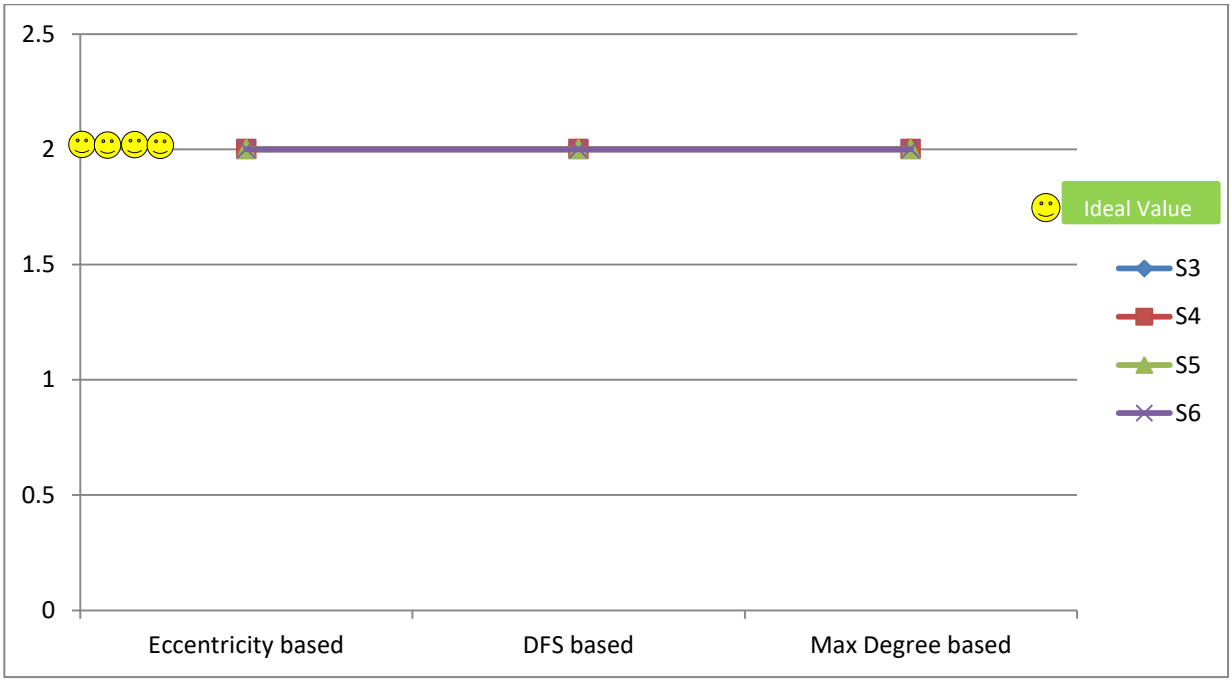
D. Star Graph (S_k – $k_{1,k}$. $\chi(G)$ – min(2, k+1))

S_3	Our Algorithm Value	Ideal / Chromatic number
Eccentricity Based	2	2
DFS Based	2	
Maximum Degree Based	2	
S_4		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
S_5		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
S_6		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	



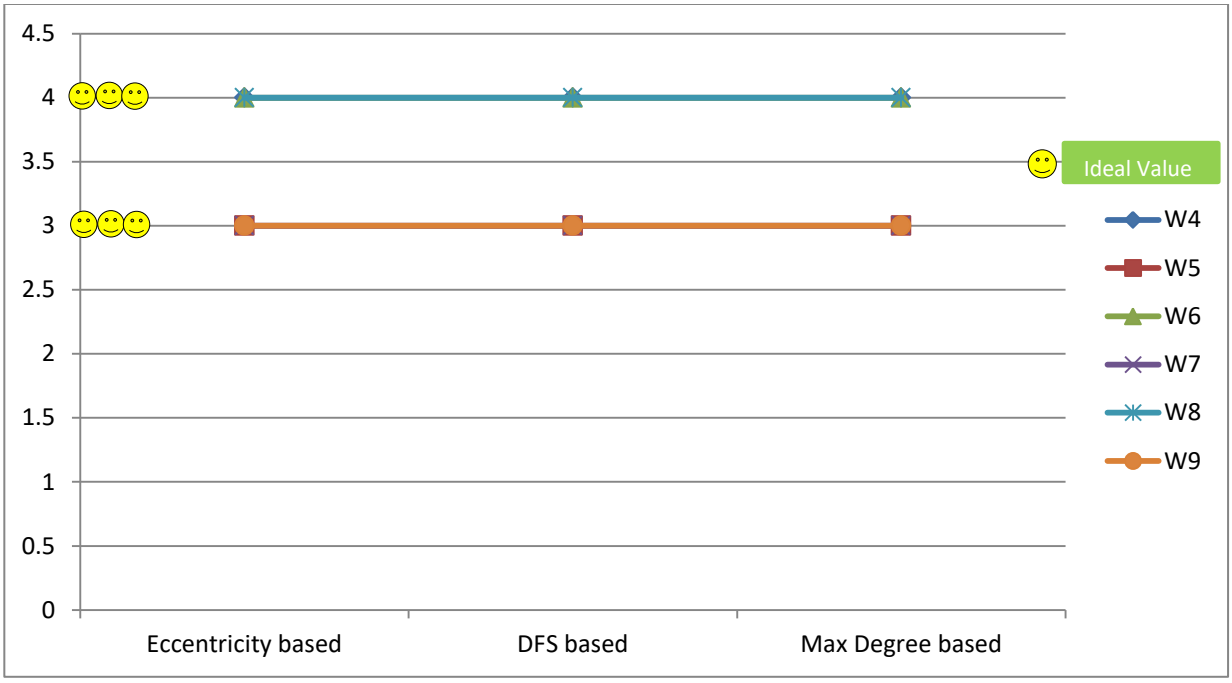
E. Path Graph (P_n)

P_2	Our Algorithm Value	Ideal / Chromatic number
Eccentricity Based	2	2
DFS Based	2	
Maximum Degree Based	2	
P_3		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
P_4		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
P_5		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	



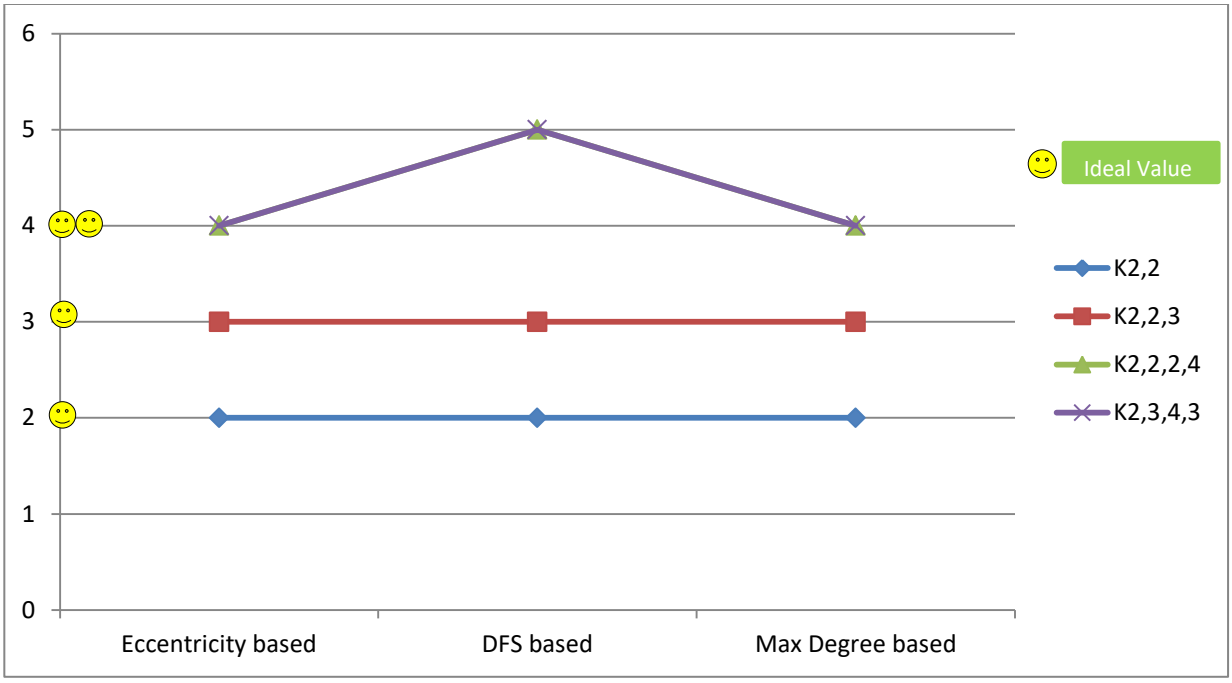
F. Wheel Graph (W_n)

W_4	Our Algorithm Value	Ideal / Chromatic number
Eccentricity Based	4	4
DFS Based	4	
Maximum Degree Based	4	
W_5		3
Eccentricity Based	3	
DFS Based	3	
Maximum Degree Based	3	
W_6		4
Eccentricity Based	4	
DFS Based	4	
Maximum Degree Based	4	
W_7		3
Eccentricity Based	3	
DFS Based	3	
Maximum Degree Based	3	
W_8		4
Eccentricity Based	4	
DFS Based	4	
Maximum Degree Based	4	
W_9		3
Eccentricity Based	3	
DFS Based	3	
Maximum Degree Based	3	



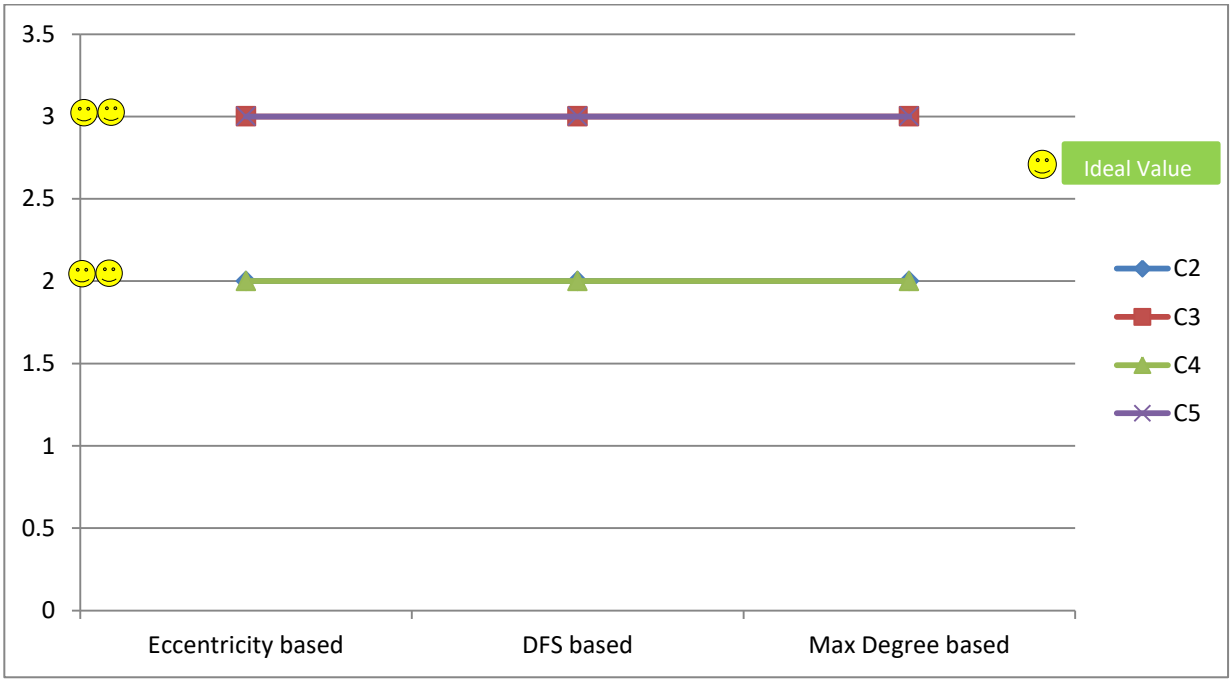
G. Complete k-partite Graph

$K_{2,2}$	Our Algorithm Value	Ideal / Chromatic number
Eccentricity Based	2	2
DFS Based	2	
Maximum Degree Based	2	
$K_{2,2,3}$		3
Eccentricity Based	3	
DFS Based	3	
Maximum Degree Based	3	
$K_{2,2,2,4}$		4
Eccentricity Based	4	
DFS Based	5	
Maximum Degree Based	4	
$K_{2,3,4,3}$		4
Eccentricity Based	4	
DFS Based	5	
Maximum Degree Based	4	


6
5
4
3
2
1
0
Eccentricity based
DFS based
Max Degree based
Ideal Value
K2,2
K2,2,3
K2,2,2,4
K2,3,4,3

G. Cycle Graph (C_n)

C_2	Our Algorithm Value	Ideal / Chromatic number
Eccentricity Based	2	2
DFS Based	2	
Maximum Degree Based	2	
C_3		3
Eccentricity Based	3	
DFS Based	3	
Maximum Degree Based	3	
C_4		2
Eccentricity Based	2	
DFS Based	2	
Maximum Degree Based	2	
C_5		3
Eccentricity Based	3	
DFS Based	3	
Maximum Degree Based	3	



CHAPTER 5
Conclusion

We have proposed and implemented new heuristic based vertex coloring algorithms. Most of our algorithms involve complementing the graph in initial or intermediate steps. The reason we used graph complement as our major tool is because complemented graph provide very close relationship among non-adjacent vertices of a graph G. Thus, it is easy to map/compute the number of independent sets. We have also freely used 'exhaustive-union' operation on the independent sets whenever required so as to reduce and obtain the number of independent sets as close as possible to the ideal chromatic value. Also, we have successfully implemented our proposed algorithms in C or object-oriented programming languages like C++ or Java according to our need and implementation requirements. We have executed our algorithms on different classes of graphs and found that all of our algorithms give exact (as ideal) value on star, path, wheel and cycle graphs. The percent accuracy values are as follows: Eccentricity based coloring has accuracy percent of 88.47, DFS based coloring has 86.25 and Maximum degree based coloring has 92.22 percent accuracy. And out of the three proposed algorithms 'Maximum Degree-based Vertex Coloring' gives the best result in terms of closeness to the ideal chromatic number. Overall, finding the new solutions to one of the world's most difficult NP-complete problem was very challenging but on the other hand was fun and rewarding as well in terms of learning.

CHAPTER 6
Future Work

The work done in this thesis can be extended to find the Chromatic Number of graph using some stochastic based techniques like 'Markov Chain Monte Carlo' based probabilistic models implemented on 'Erdos-Renyi Random Graph Generator'.

References

[1] New methods to color the vertices of graph. *Communications of ACM*, Volume 22 Issue 4, 1979.

[2] Graph Coloring Algorithms. *Graph Theory and Computing*, pages 109–122, 1972.

[3] Introduction to Graph Theory. *Douglas B. West*, Second edition.

[4] Heuristic Methods for Graph Coloring Problems. *ACM Symposium on Applied Computing*, 2005.

[5] Link: https://heuristicswiki.wikispaces.com/Graph+coloring

[6] Link: https://en.wikipedia.org/wiki/Graph_coloring

[7] Link: https://mathworld.wolfram.com/wiki/ChromaticNumber.html

[8] Harary 1962; Skiena 1990, p. 179; West 2000, p. 151
http://mathworld.wolfram.com/HararyGraph.html

Appendix

Code: Eccentricity Based Vertex Coloring

```
31                      /* Writing helper function */
32                      string get_color_string(int color,int max_color); // TODO(brrcrites): remove w/ dotty removal
33                      int find_max_color();
34
35              public:
36                      /* Constructors */
37                      GraphColor(map<string, vector<string> > input_graph): graph(input_graph), colored(false) {}
38
39                      /* Mutators */
40                      virtual void set_condition(int con) = 0;
41                      virtual map<string,int> color() = 0;
42                      void set_graph(map<string,vector<string> > new_graph) { this->graph = new_graph; this->colored = false; }
43                      void modify_graph(string node, vector<string> neighbors) { this->graph[node] = neighbors; this->colored = false; }
44                      void set_coloring(map<string, int> coloring) { this->coloring = coloring; }
45                      virtual bool verify();
46
47                      /* Accessors */
48                      virtual string get_algorithm() = 0;
49                      unsigned size() { return this->graph.size(); }
50                      bool is_colored() { return this->colored; }
51                      map<string,int> get_coloring() { return this->coloring; }
52                      int get_color(string node);
53
54                      /* Print functions */
55                      void print_coloring();
56                      void print_chromatic();
57                      void write_graph(string graph_name = ""); // TODO(brrcrites): remove w/ dotty removal
58      };
59  }
```

```
35              public:
36                      /* Constructors */
37                      GraphColor(map<string, vector<string> > input_graph): graph(input_graph), colored(false) {}
38
39                      /* Mutators */
40                      virtual void set_condition(int con) = 0;
41                      virtual map<string,int> color() = 0;
42                      void set_graph(map<string,vector<string> > new_graph) { this->graph = new_graph; this->colored = false; }
43                      void modify_graph(string node, vector<string> neighbors) { this->graph[node] = neighbors; this->colored = false; }
44                      void set_coloring(map<string, int> coloring) { this->coloring = coloring; }
45                      virtual bool verify();
46
47                      /* Accessors */
48                      virtual string get_algorithm() = 0;
49                      unsigned size() { return this->graph.size(); }
50                      bool is_colored() { return this->colored; }
51                      map<string,int> get_coloring() { return this->coloring; }
52                      int get_color(string node);
53
54                      /* Print functions */
55                      void print_coloring();
56                      void print_chromatic();
57                      void write_graph(string graph_name = ""); // TODO(brrcrites): remove w/ dotty removal
58      };
59  }
60
61  #endif // _GRAPH_COLOR_H_
```

```
#ifndef _GRAPH_COLOR_H_
#define _GRAPH_COLOR_H_

#include <map>
#include <stack>
#include <locale>
#include <vector>
#include <string>
#include <utility>
#include <iostream>
#include <queue>

using std::queue;
using std::map;
using std::pair;
using std::stack;
using std::string;
using std::vector;
using std::cout;
using std::cerr;
using std::endl;

namespace GraphColoring {
        class GraphColor {
                protected:
                        /* Internal Members */
                        map<string,vector<string> > graph;
                        map<string,int> coloring;
                        bool colored;
```

Code: DFS Based Vertex Coloring

```
                return this->coloring.at(node);
        }
        return -1;
}

// Used to print the Chromatic Color
void GraphColoring::GraphColor::print_chromatic() {
    int largest = -2;
    for(map< string, int >::iterator itr = this->coloring.begin(); itr != this->coloring.end(); itr++) {
        if(itr->second > largest) {
            largest = itr->second;
        }
    }
    cout << this->get_algorithm() << " Chromatic Number: " << largest+1 << endl;
}

// Used to print the color of each node in the graph
void GraphColoring::GraphColor::print_coloring() {
    std::cout << "----------" << this->get_algorithm() << " Colorings----------" << endl;
    for(map< string,int >::iterator itr = coloring.begin(); itr != coloring.end(); itr++) {
        std::cout << itr->first << " " << itr->second << endl;
    }
}

int GraphColoring::GraphColor::find_max_color() {
    map<string,int>::iterator color_it = coloring.begin();
    int max_color = 0;
    for (/*color_it*/;color_it != coloring.end();color_it++) {
        if ((*color_it).second > max_color) {
```

```
    ofstream outfile(filename.c_str());
    if (!outfile.is_open()) {
        cerr << "Error: Unable to open \"" << filename << "\"." << endl;
        return;
    }
    outfile << "graph " << graph_name << " {\n";
    map <string,vector<string> >::iterator graph_it;
    graph_it = graph.begin();
    for (/*graph_it*/;graph_it != graph.end();graph_it++) {
        outfile << (*graph_it).first << "[label=\"" << (*graph_it).first << "\\n"
                << coloring.at((*graph_it).first) << "\"";
        if (colored) {
            outfile << " style=filled fillcolor=\"";
            outfile << get_color_string(coloring.at((*graph_it).first),max_color);
            outfile << "\"";
        }
        outfile << "];\n";
    }
    graph_it = graph.begin();
    for (/*graph_it*/;graph_it != graph.end();graph_it++) {
        string start_node = (*graph_it).first;
        vector<string> connections = (*graph_it).second;
        for (unsigned i = 0;i < connections.size();i++) {
            if (connections.at(i) < start_node) {
                outfile << start_node << " -- " << connections.at(i) << endl;
            }
        }
    }
    outfile << "}" << endl;
```

```
            max_color = (*color_it).second;
        }
    }
    return max_color;
}

#include "../Header/graph_colors.h"

// TODO(brrcrites): remove the next two functions w/ dotty removal
string GraphColoring::GraphColor::get_color_string(int color,int max_color) {
    return ColorArray[color];
    const int MaxColor = 1023;
    color = color * (MaxColor / max_color);
    std::stringstream color_changer;
    color_changer << "0x" << std::hex << color;
    return color_changer.str();
}

void GraphColoring::GraphColor::write_graph(string graph_name) {
    int max_color = find_max_color();
    if(max_color > COLOR_ARRAY_SIZE)
    {
        cerr << "Error: Graph has too many colors to be written" << endl;
        return;
    }
    if (graph_name.empty()) {
        graph_name = "colored_graph";
    }
    string filename = graph_name+".dot";
```

```
        return;
    }
    outfile << "graph " << graph_name << " {\n";
    map <string,vector<string> >::iterator graph_it;
    graph_it = graph.begin();
    for (/*graph_it*/;graph_it != graph.end();graph_it++) {
        outfile << (*graph_it).first << "[label=\"" << (*graph_it).first << "\\n"
                << coloring.at((*graph_it).first) << "\"";
        if (colored) {
            outfile << " style=filled fillcolor=\"";
            outfile << get_color_string(coloring.at((*graph_it).first),max_color);
            outfile << "\"";
        }
        outfile << "];\n";
    }
    graph_it = graph.begin();
    for (/*graph_it*/;graph_it != graph.end();graph_it++) {
        string start_node = (*graph_it).first;
        vector<string> connections = (*graph_it).second;
        for (unsigned i = 0;i < connections.size();i++) {
            if (connections.at(i) < start_node) {
                outfile << start_node << " -- " << connections.at(i) << endl;
            }
        }
    }
    outfile << "}" << endl;
    outfile.close();
}
```

```
    }
    return split_string;
}

vector< vector<string> > get_input(char* input_file) {
    vector< vector<string> > Input;

    ifstream file(input_file);
    if(file.is_open()) {
        string line;
        while(Getline(file,line)) {
            vector<string> words = split(line);
            if(words.size() > 1) {
                for(unsigned i=1; i<words.size(); i++) {
                    if(words[i] != "0" && words[i] != "1" && words[i] != "x" && words[i] != "X") {
                        cerr << "Problem with this Input Line: " << line << endl;
                        cerr << "Problem is with word: \"" << words[i] << "\" at position " << i << endl;
                        Input.clear();
                        return Input;
                    }
                }
            } else {
                cerr << "No Actuation Data Found, Please Check Input File" << endl;
                Input.clear();
                return Input;
            }
            Input.push_back(words);
        }
```

Code: Maximum Degree Based Coloring

```
    GraphColor *graph = new HybridDsatur(input_graph);


    graph->color();
    graph->print_chromatic();
    graph->verify();
    graph->write_graph();

    return 0;
}

ifstream& Getline(ifstream& ifs,string& line) {
    getline(ifs,line);
    if (!line.empty()) {
        while (isspace(line.at(line.size()-1))) {
            line = line.substr(0,line.size()-1);
        }
    }
    return ifs;
}

vector<string> split(string to_split) {
    vector<string> split_string;
    unsigned index_start;
    for (unsigned i = 0;i < to_split.length();i++) {
        index_start = i;
        while(i < to_split.length() && !isspace(to_split.at(i))) { i++; }
        split_string.push_back(to_split.substr(index_start,i - index_start));
```

```
        }
        for(int i=0; i<vertices; i++) {
            string pre = "v";
            string temp;
            std::ostringstream convert;
            convert << (i+1);
            temp = convert.str();
            pre.append(temp);
            vector<string> base;
            input_graph[pre] = base;
        }
        while(Getline(file,line))
        {
            vector<string> words = split(line);
            if(words[0] == "e")
            {
                string arg1 = "v";
                arg1.append(words[1]);
                string arg2 = "v";
                arg2.append(words[2]);
                vector<string> base;
                vector<string> base2;
        input_graph[arg1].push_back(arg2);
        input_graph[arg2].push_back(arg1);
            }
        }
    } else {
        cerr << "Input File Not Found" << endl;
```

```
        return;
    }
}

//Used to parse test inputs where the first line is the number of
//vertices, and the next lines are the edge matrix
void parse_edge_matrix(char* input_file) {
    string pre = "v";

    ifstream file(input_file);
    if(file.is_open()) {
        string line;
        Getline(file,line);
        int n = atoi(line.c_str());
        int i = 0;
        while(Getline(file,line)) {
            i += 1;
            vector<string> words = split(line);
            if((int)words.size() != n) {
                cerr << "Invalid Input, line " << i << " is not the correct length ("
                     << words.size() << "," << n << "): " << line << endl;
                for (unsigned i = 0;i < words.size();i++) { cerr << "\t" << words.at(i) << endl; }
                input_graph.clear();
                return;
            }
```

```
            vector<string> edges;
            for(int j = 0; j < n; j++) {
                if(words[j] == "1") {
                    pre = "v";
                    string temp;
                    ostringstream convert;
                    convert << (j+1);
                    temp = convert.str();
                    edges.push_back(pre.append(temp));
                }
            }
            pre = "v";
            string temp;
            ostringstream convert;
            convert << i;
            temp = convert.str();
            input_graph[pre.append(temp)] = edges;
        }
        if(i != n) {
            cerr << "Input is not the right length" << endl;
            input_graph.clear();
            return;
        }
        file.close();
    } else {
        cerr << "Input File Not Found" << endl;
        return;
```

```
        file.close();
    } else {
        cerr << "Input File Not Found" << endl;
    }
    return Input;
}

void parse_edge_list(char* input_file) {
    ifstream file(input_file);
    if(file.is_open()) {
        string line;
        int vertices = -1;
        int flag = 0;
        while(!flag && Getline(file,line)) {
            while(line.size() == 0) {
                Getline(file,line);
            }
            vector<string> words = split(line);
            if(words.size() != 0) {
                if(words[0] == "p") {
                    vertices = atoi(words[2].c_str());
                    flag = 1;
                }
            }
        }
        if(!flag || vertices == -3) {
            cerr << "File is missing parameter line before edge list" << endl;
            cerr << "Should be: \"p edge <number of vertices> <number of edges>\"" << endl;
            return;
```