Splittable and Highly Connectable Degree Sequences

by

Shreyas Mavani
202111008

A Thesis Submitted in Partial Fulfilment 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, 2023

Declaration

I hereby declare that

i) The thesis comprises 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 else where for a degree,

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

[illegible]

Shreyas Mavani

Certificate

This is to certify that the thesis work entitled Splittable and Highly Connectable Degree Sequences has been carried out by Shreays Mavani 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

Prof. Rahul Muthu
Thesis Supervisor

Acknowledgments

First and foremost, I would like to thank my mother, Mrs. Ritaben Mavani and sister Komal Mavani for always supporting me and believing that I can achieve what I desire. My father, Mr. Satishbhai Mavani has always instilled the essence of competing and evolving as a better professional. I would also like to thank my grandparents, who always showered blessings on me.

I want to express my heartfelt gratitude to my thesis guide, Prof. Rahul Muthu, for his invaluable guidance and support throughout the research work.His guidance, knowledge, and expertise have helped me achieve this research's objectives.

I would also like to thank Prof. Anuj Tawari for his guidance and support in completing this research. His valuable input and suggestions helped refine my work and improve its quality.

I am grateful to my panel members Prof. V. Sunitha, Prof. Saurish Das Gupta,Prof. Mukesh Tiwari, Prof. Priyanka Singh, and Prof. Sanjay Shrivastav, for their valuable time and insightful feedback during my stage presentations. I want to thank all the professors at DA-IICT who have contributed to my evolution. Their teaching provided good knowledge, which was helpful during the thesis work, and will be there with me as I grow.

I am grateful to my friends of ML family who helped me directly or indirectly emotionally, mentally,or physically in this journey.

Contents

Abstract

This work explores the broad domain of the relationship between graphs with the same degree sequence, aiming to find a degree sequence characterization for certain graph classes. Degree sequences represent the sequence of degrees of vertices in a graph. While degree sequences alone do not guarantee the structure of a graph, they can provide a foundation for further development and offer conditions that can be tested in linear time.

Earlier work in this domain includes graph classes that are degree determined, meaning that their membership can be determined solely based on the degree sequence. Examples of such classes include complete graphs, split graphs, and threshold graphs. Linear time algorithms exist for recognizing whether a degree sequence belongs to these classes. The concept of 2-switch operations was introduced, which involve edge deletion and addition to change the adjacencies in a graph. Performing a 2-switch operation may change the isomorphism class of the graph. The configurations in a 2-switch operation that lead to a change in the isomorphism class have been studied. Realization graphs are used to study the relationship between different realizations of a degree sequence. It was shown that realization graphs are always connected, as a consequence of the Fulkerson, Hoffman, and McAndrew theorem.

In this thesis Our contribution to this domain involves solving two specific problems.These are : study of disconnected realizations and highly connected realizations of degree sequences. The definitions and notation used throughout the work are presented, including the definitions of degree sequence, graphic sequence, unigraphic degree sequence, and the Havel-Hakimi construction method.

The abstract provides an overview of the study's focus on degree sequences, graph classes, 2-switch operations, realization graphs, and their properties.

List of Figures

CHAPTER 1

Introduction

Degree sequence is one of the simplest terms associated with a graph. However, as most of the degree sequences belong to several pairwise non-isomorphic graphs, the use of degree sequences in graph problems is limited since the degree sequence does not guarantee the structure of a graph. Hence it is desirable to find out the relationship between graphs with the same degree sequence. The motivation for this work was to study various classes of graphs that can be stated in terms of their degree sequence characterization that will provide us with some foundation for further development. Also, such characterization is important because conditions depending only on the degree sequence can often be tested in linear time.

A graph family is said to be degree determined if by knowing only the degree sequence, we can determine whether the graph belongs to it or not. Most of the graph classes are not degree determined. However, some are. For example complete graphs, split graphs, and threshold graphs. Each of these graph classes has a linear time algorithm for recognizing whether a degree sequence belongs to it or not. We seek to characterize these degree sequence-determined classes using 2-switch operations. A 2-switch is an edge deletion and addition operation which changes the adjacencies in the graph, as a result changing the isomorphism class of the graph in many cases. We have also studied the configurations in a 2-switch operation that will change the isomorphism class of the graph.

The relationship that exists between various realizations of a degree sequence can be very well studied with the help of realization graphs. The best-known result of a realization graph is that it is a connected graph for any degree sequence d; this is a consequence of a theorem of Fulkerson, Hoffman, and McAndrew. This naturally motivates us to find the characterization of such a graph.

1.1 Problem Statement

Study properties of disconnected realizations as well as high connectivity realizations of degree sequences.

1.2 Definitions and Notation

In this chapter, we present some definitions and notation which we will be using throughout the thesis.

1.2.1 Degree Sequence

1.2.1.1 Definition

Given an undirected graph G on n vertices, a degree sequence d = $(d_1, d_2, \ldots, d_n)$ is a monotonically non-increasing sequence of the vertex degrees[9]

1.2.1.2 General Characteristics of Degree Sequences

The sum of the elements of a degree sequence of a graph is always even due to the fact that each edge connects two vertices and is thus counted twice.[2]

$$\sum d(i) = 2 \times number\_of\_edges\_in\_graph \tag{1.1}$$

Two graphs with different degree sequences cannot be isomorphic. For example, the following two graphs are not isomorphic,

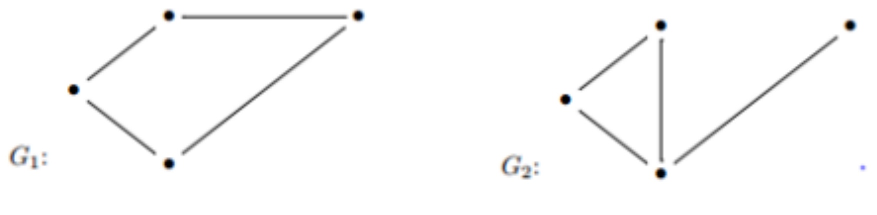


Figure 1.1: Non-Isomorphic Graphs with Different Degree Sequences

Here, G_1 and G_2 have degree sequences (2, 2, 2, 2) and (1, 2, 2, 3) respectively. Since G_1 one has four vertices of degree 2 and G_2 has only two vertices with degree 2. So we can say that G_1 and G_2 can never be isomorphic to each other.

However, having the same degree sequence is not sufficient for the isomorphism of graphs. Even simple connected graphs with the same degree sequences can be non-isomorphic. For example,

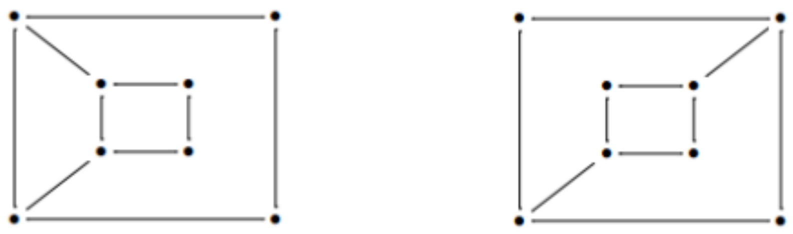


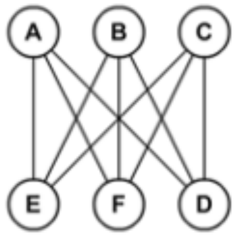
Figure 1.2: Non-Isomorphic Graphs with Same Degree Sequences

1.2.2 Graphic Sequence

1.2.2.1 Definition

A degree sequence d = $(d_1, d_2, \ldots, d_n)$ is said to be graphic if d is the degree sequence of some graph G = (V, E) on the vertex set V = $\{1, 2, \ldots, n\}$.such that deg(i) = (d_i) for all i.[8]

In this case, we say that G realizes d or is a realization of d. While it is easy to determine if a sequence is graphic, a given graphic sequence can have many different realizations. It is possible that the realizations of the same graphic sequences are nonisomorphic to each other. So it is important to the study properties of such graphic sequences and their overlap with various classes of graphs.



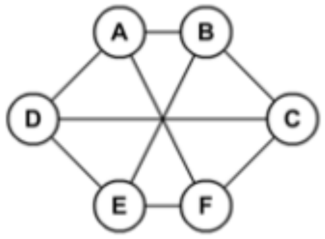


Figure 1.3: Different Realizations of same degree sequence (3,3,3,3,3,3)

1.2.2.2 Unigraphic Degree Sequence

A degree sequence d is said to be unigraphic if and only if it is uniquely realizable i.e. only one realization can be obtained from d. In order words, d is said to be uniquely realizable. For example, complete graphs are uniquely realizable. The graph family of a unigraphic degree sequence can be determined in linear time[8].

1.2.2.3 Havel-Hakimi construction method

Havel-Hakimi has given a systematic approach for generating a realization from a given non-increasing degree sequence d $=(d_1, d_2, \ldots, d_n)$.

1. Select vertex with highest degree (d_1) and connect it to next $|d_1|$ followed vertices in degree sequence and decrement degree of adjacent vertices by 1.

2. If degree sequence contains all 0's then terminate else using stable sort rearrange degree sequence into non-increasing order and repeat step 1.[4]

1.3 2-switch

1.2.3.1 Definition of an alternating 4-cycle

An alternating 4-cycle is an instance of four vertices a, b, c, d in a graph G such that ab and cd are edges in G and bc and ad are not present in G. We denote this alternating 4-cycle by < a,b:c,d >.[1]

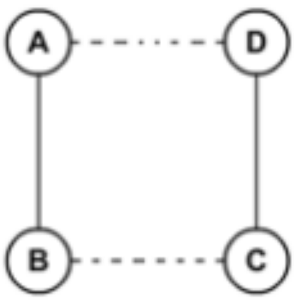


Figure 1.4: An alternating 4-cycle

1.2.3.2 Definition of a 2-Switch

The 2-switch < ab:cd > is the operation of deleting edges ab and cd from and adding edges ad and bc to G. (This operation has often also been called a transfer or cycle exchange). A 2-switch on < a,b:c,d > is as shown below.

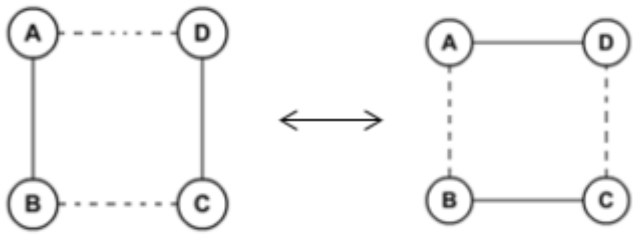


Figure 1.5: 2-Switch operation on < a,b:c,d >

Note that performing a 2-switch is an operation that changes the adjacencies in the graph while preserving the degree of each vertex since the edges are deleted and added at the same vertices. Hence, a 2-switch on G results in a graph G′ in which every vertex has the same degree as it had before; thus G′ is another realization of the degree sequence of G. Since the 2-switch operation changes the adjacencies in the graph, the new realization obtained after performing the 2-switch may be non-isomorphic to the previous graph. Also as the 2-switch operation makes only local changes, the rest of the graph remains unaffected.

1.2.3.3 Example

Performing a 2-switch operation on < 1,5 : 3,4 > in the below figure:

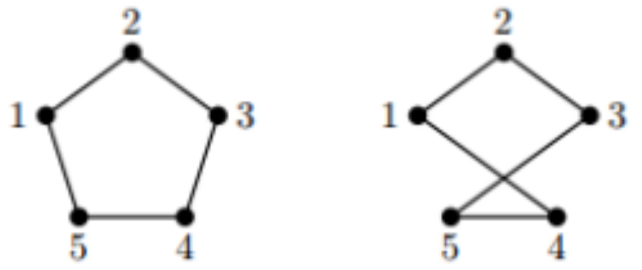


Figure 1.6: Two realizations of (2, 2, 2, 2, 2) differing by a single 2-switch

1.2.3.4 Fulkerson Theorem

A well-known result of Fulkerson, Hoffman, and McAndrew links graphs having the same degree sequence.

Theorem 1.1: Two unlabeled graphs G and H have the same degree sequence if and only if there is a sequence of 2-switches that transforms G into H.[10]

Hence, it is possible to generate all possible realizations of a degree sequence using 2-switch operations.

1.2.3.5 Reversibility of 2-switch

A 2-switch operation is reversible because if given its output graph, it is always possible to determine back its input graph. Suppose a 2-switch < a,b:c,d > is performed on graph G and the output graph is G′, then graph G can be obtained back from G′ by performing the following 2-switch on G′: < a,d:c,b >[1].

1.3.1 Degree Sequence Characterization

A graph class C is said to be degree determined, or that it has a degree sequence characterization if it is possible to determine whether a graph G belongs to C from just the degree sequence of G.

1.3.2 Closure under 2-switch

A graph class C is said to be closed under a 2-switch operation if and only if performing a 2-switch on any graph G belonging to C results in a graph G′ also belonging to C. In other words, performing a 2-switch on the graphs of class C produces a graph of the same class C.

1.3.3 Realization Graph

The realization graph G(d) is the graph (R, δ), where R is the set of realizations of d, and two vertices G, G′ of R are adjacent if and only if performing some 2-switch changes G into G′. Since the operation of undoing a 2-switch is itself a 2-switch, G(d) may be thought of as an undirected graph. The best-known result on G(d) is that for any degree sequence d, its realization graph is connected; this is a consequence of a theorem of Fulkerson, Hoffman, and McAndrew [Theorem 1.1].[3]

1.3.4 Minimal edge cut

The minimal edge cut in a graph refers to the smallest set of edges that, when removed from the graph, disconnects it into two or more separate components. This cut divides the graph into two or more disjoint subgraphs.

1.3.5 Perfect Matching

Matching refers to a subset of edges in a graph such that no two edges share a common vertex. In other words, a matching is a set of edges where each vertex is incident to at most one edge in the set.

A perfect matching is a matching that covers all the vertices in a graph. In other words, every vertex is incident to exactly one edge in the matching. Not all graphs have perfect matchings, and determining whether a graph has perfect matching is a well-studied problem.[10]

1.4 Max Flow

Maximum flow is a fundamental concept in graph theory that represents the maximum amount of flow that can be sent through a network from a source node to a sink node. It has various applications, including network optimization, transportation planning, and resource allocation.

The maximum flow problem aims to find the optimal flow configuration that maximizes the amount of flow from the source to the sink while respecting the capacities of the edges in the network. The Ford-Fulkerson algorithm, which has different variations such as the Edmonds-Karp algorithm, is commonly used to solve the maximum flow problem.

1.5 Min Cut

In graph theory, a minimum cut (also known as a min-cut) refers to the smallest possible cut that can be made in a graph to disconnect it into two separate components. The cut consists of a set of edges whose removal would result in two disjoint subgraphs.

Finding the minimum cut in a graph is a well-studied problem with various algorithms available. One common algorithm to solve this problem is the Ford-Fulkerson algorithm, specifically the variant known as the Edmonds-Karp algorithm, which is based on augmenting paths.

CHAPTER 2
Literature Survey

In this chapter, we have mentioned various properties and configurations related to 2-switches.

2.1 Conditions for 2-switch

Theorem 2.1: Given a graph G, a 2-switch operation can be performed in G, if and only if an induced sub-graph on any 4 vertices of G lies in one of these three configurations: $2K_2$, P_4, C_4.

So, The presence of an alternating 4-cycle leads to the 2-switch operation. Hence, four vertices are required to perform a 2-switch, also at least one perfect matching must be present and one perfect matching must be absent

Following are the only three configurations ($2K_2$, P_4, C_4) on four vertices in which a 2-switch can be applied:

1. $2K_2$

   In $2K_2$ since only one perfect matching is present and the rest four edges are absent; two distinct 2-switches are possible i.e. < A,B : C,D > and < A,B : D, C > (as shown in the below figure). In this case, the result after performing a 2-switch is also a perfect matching i.e. $2K_2$.

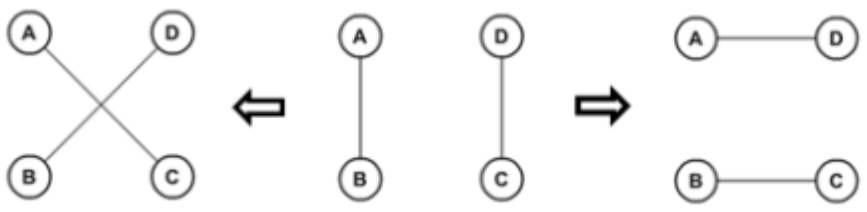


Figure 2.1: Two different possible 2-switches in $2K_2$

2. P_4

In P_4, only a single 2-switch operation is possible i.e. < A,B:D,C > (as shown in the below figure). The result is also a P_4

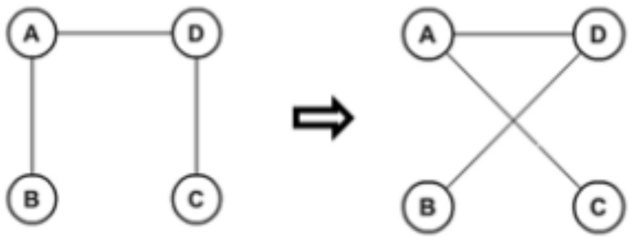


Figure 2.2: Only one possible 2-switch in P_4

3. C_4

In C_4 since two perfect matching are present and rest two edges are ´ absent; two distinct 2-switches are possible i.e.< A,B:D,C > and < A,D:B,C > as shown in the below figure. In this case, the result after performing a 2-switch is also a C_4

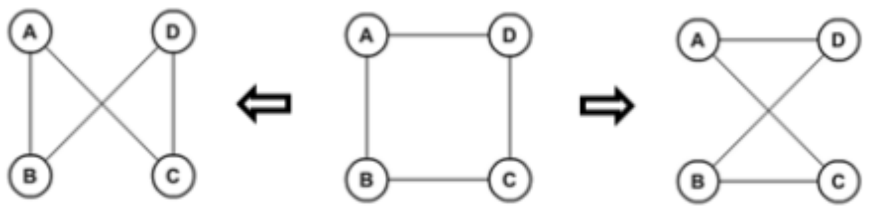


Figure 2.3: Two different possible 2-switches in C_4

Also, there are no other graphs on four vertices on which the 2-switch can be applied. Hence, in general, to perform a 2-switch on graph G, the induced sub-graph on any four vertices must be in one of this three configurations:$2K_2$, P_4 and C_4.

Hence, in graph G, if there is no induced sub-graph on four vertices containing $2K_2$, P_4, and C_4 then we can say that a 2-switch cannot be performed on G.

For a given graph there exists a unique degree sequence but from a given degree sequence there exists, in general, more than one graph which is not isomorphic to each other. Using a 2-switch operation it is possible to generate all possible realizations of a degree sequence. However the application of a single 2-switch might change the isomorphism class of the graph. So it is necessary to study the configurations in which a 2-switch operation changes the isomorphism class of the graph.

2.2 2-switch and Isomorphism

We give a necessary condition that if a 2-switch changes the isomorphism class of a graph, then it must take place in one of four configurations. We also present a sufficient condition for a 2-switch to change the isomorphism class of a graph.

Since a 2-switch changes the adjacencies in the graph, intuitively one can say that it will most often change the isomorphism class of the graph. However, this is not always the case. A 2-switch operation may not change the isomorphism class of the graph. For example, performing a 2-switch on P_4 will result in another P_4.

2.2.1 Necessary condition for changing isomorphism class

Theorem 2.2 : If graph G admits a 2-switch that changes the isomorphism class of the graph, then G contains one of the configurations in Figure 2.4, with vertices p, q, r, s as marked, such that the 2-switch is on < p,q:r,s >.

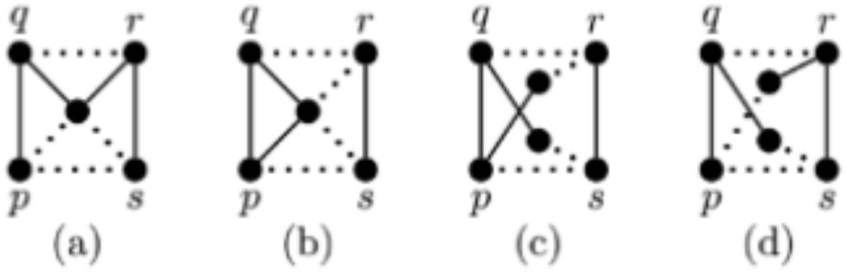


Figure 2.4: Configuration for isomorphism class changing 2-switches

2.2.2 Sufficient condition for changing isomorphism class

Theorem 2.3: If graph G contains one of the configurations shown in Figure 2.4, and the 2-switch on < p,q:r,s > produces a graph isomorphic to G, then the vertices of the configuration lie in one of the configurations in Figure 2.5, where the alternating cycle < u,v:w,x > coincides with < p,q:r,s >.

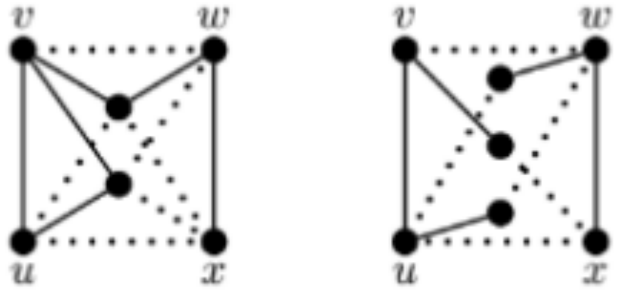


Figure 2.5: Configuration for isomorphism class preserving 2-switches

2.3 Closure under 2-switch

A graph class C is said to be closed under a 2-switch operation if and only if performing a 2-switch on any graph G belonging to C results in a graph G′ also belonging to C. In other words, performing a 2-switch on the graphs of class C produces a graph of the same class C. If performing a 2-switch operation on a graph G belonging to graph class C changes the graph class of the new graph to C′ , then we can say that both C and C′ are not closed under the 2-switch operation.

1. Complete Graph

   A complete graph is a simple undirected graph in which every pair of distinct vertices are connected by a unique edge.

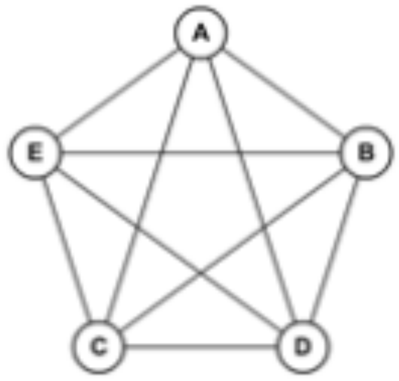


Figure 2.6: Complete graph on 5 vertices

   In the case of complete graphs, an induced sub-graph on any four vertices will always be complete since all vertices are connected to each other. As there is no alternating cycle in a complete graph, no 2-switch operation can be performed. Hence, we can say that complete graphs are degree determined.

2. Split Graph A split graph G is a graph in which the vertex set can be partitioned into an independent set I and a clique C. A split graph can have more

than one partition to a clique and an independent set.

Split graphs are closed under a 2-switch operation.

Case i: All four vertices of the 2-switch are either selected from set I or all are selected from set C.

As seen in the above section, no 2-switch operation can be performed in a complete graph. Also, an independent set is an empty graph, so a 2-switch cannot be performed on the vertices of an independent set. So, in the case of split graphs, to perform a 2-switch we cannot select all four vertices from the independent set (or clique).

Case ii: Three vertices are selected from set I and one from set C or three vertices are selected from set C and one from set I

In both cases, two switch is not possible due to the absence of an alternating 4-cycle (Theorem 2.1).

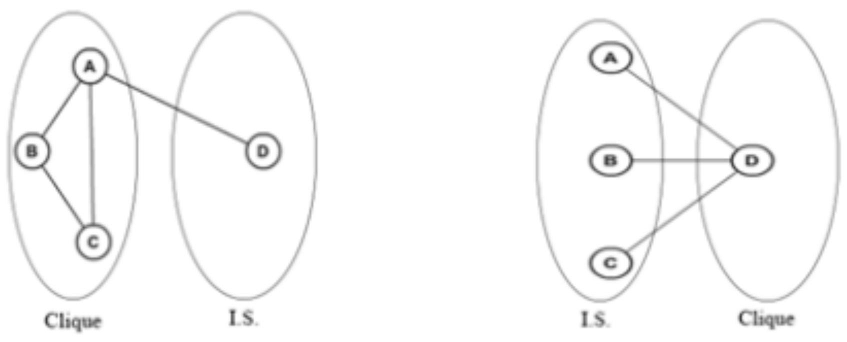


Figure 2.7: Absence of an alternating 4 cycle on selecting three vertices from the same set

Case iii: Two vertices are selected from set I and two from set C

Here, two vertices from a set I will be non-adjacent to each other as they lie in an independent set and the two vertices from set C will be adjacent to each other because they lie in a clique. The only configuration in which a 2-switch can be performed in a split graph is shown in the figure below.

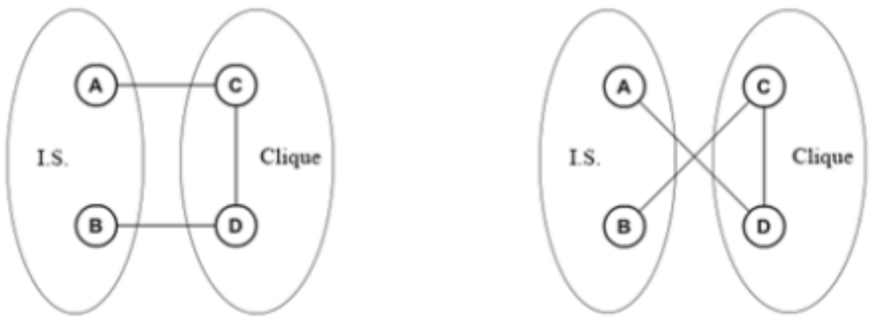


Figure 2.8: 2-switch configuration in split graph

As we can see, the 2-switch only changes the crossing edges between the independent set and the clique, the rest is unchanged. Hence, we can conclude that the resultant graph is also a split graph.

3. Planar Graph

A planar graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other. Such a drawing is called a plane graph or planar embedding of the graph.

Planar graphs are not closed under 2-switch operation i.e. are not degree determined. For example,

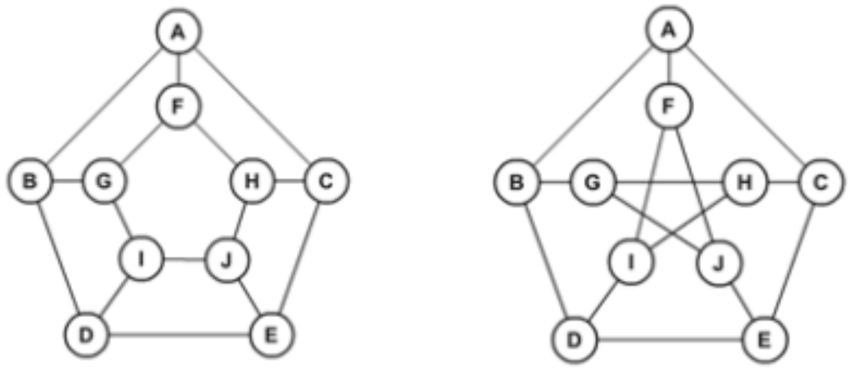


Figure 2.9: Planar and Non-Planar drawing of (3^{10})

4. Bipartite graph

   A simple graph G = (V, E) with vertex partition V = V_1, V_2 is called a bipartite graph if every edge of E joins a vertex in V_1 to a vertex in V_2. In general, a Bipartite graph has two sets of vertices, let us say, V_1 and V_2, and if an edge is drawn, it should connect any vertex in set V_1 to any vertex in set V_2.

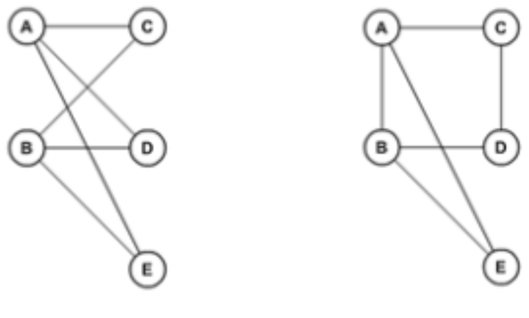
   

   Figure 2.10: 2-switch on $K_{2,3}$

   Bipartite graphs are not closed under a 2-switch operation since the adjacencies are changed due to which bi-partition may not be possible due to the formation of odd cycles. The above figure shows the class changing 2-switch on $K_{2,3}$. Hence, bipartite graphs are not degree-determined classes of graphs.

5. Threshold Graph

A threshold graph is a graph that can be constructed from a single vertex graph by repeated application of the following two operations

(a) Addition of a single isolated vertex to the graph.

(b) Addition of a single dominating vertex i.e. a single vertex that is connected to all other vertices in the graph.

A threshold graph is a graph with no induced $2K_2$, P_4 or C_4. Hence, by Theorem 2.1, we can say that a 2-switch operation cannot be performed on the threshold graph. So, threshold graphs are degree determined.

CHAPTER 3
Splittable Sequence

A splittable sequence refers to a sequence or a list that can be divided or split into multiple smaller sequences while preserving the original order of elements. In other words, a splittable sequence allows you to break it apart into smaller sub-sequences without rearranging or altering the order of the elements.

For example, consider the sequence [1, 2, 3, 4, 5, 6]. This sequence can be split into two smaller sequences: [1, 2, 3] and [4, 5, 6]. Each sub-sequence retains the order of the elements from the original sequence.

Splitting a sequence can be useful in various scenarios, such as dividing a large dataset into smaller batches for processing, partitioning a list for parallel computation, or organizing data into manageable chunks for analysis or manipulation.

3.1 Havel Hakimi Algorithm

3.1.1 theorem

For n>1, an integer list d of size n is graphic if and only if d′ is graphic, where d′ is obtained from d by deleting its largest element X and subtracting 1 from its X next largest elements. where The only 1-element graphic sequence is 0.[5] [6]

3.1.2 Method

The Havel-Hakimi algorithm is a mathematical algorithm used for solving certain problems related to graph theory and sequence theory. Specifically, it is used to determine if a given sequence of non-negative integers is graphical, meaning it can represent the degree sequence of a simple undirected graph.

Here is a step-by-step explanation of the Havel-Hakimi algorithm:

1. Start with a sequence of non-negative integers. Sort the sequence in non-increasing order.
2. Check if the sequence is empty. If it is, then the original sequence is graphical.
3. Check if the first element of the sorted sequence is negative or greater than the length of the sequence minus one. If either of these conditions is true, then the original sequence is not graphical.
4. If the first element is zero, remove it from the sequence.
5. Subtract 1 from the first element of the sequence.
6. Repeat steps 2 to 5 with the modified sequence obtained in step 5.

By following these steps iteratively, you either arrive at an empty sequence, indicating that the original sequence is graphical, or you encounter a situation where the sequence fails one of the conditions, indicating that the original sequence is not graphical.

The Havel-Hakimi algorithm relies on the fact that a sequence is graphical if and only if a modified sequence obtained from it is also graphical. By reducing the largest element in each step and checking the conditions, the algorithm progressively simplifies the sequence until it reaches a clear conclusion.

3.2 Problem on Splittable Sequence

1. Given a degree sequence, we have to split this sequence into two-degree sequences.
2. We first partition the input sequence with one part for each distinct degree.
3. We split each such part into two: part 1 and part 2.
4. Merge all parts 1 in one sequence and all parts 2 in another sequence. Then check if it is a valid break or not.
5. Then count the number of valid breaks.

3.3 Implementation using Dynamic Programming

First, the Degree sequence of n vertices in non-increasing order.

In this sequence we have k different degree $(d_1, d_2, \ldots, d_k)$ with its frequency $(f_1, f_2, \ldots, f_k)$ where

1. $(f_1 + f_2 + \ldots + f_k)$= n
2. $f_i > 0$
3. $d_i > 0$.

Now, we have to divide frequencies into two parts of degree sequence like l_1 to l_k and r_1 to r_k Where for every k, f_i - l_i = r_i.

So, now we have to check both parts of the degree sequence using Havel-Hakimi Algorithm for the valid split.

So we got total (f_1+1)*(f_2+1)........*(f_k+1) various no of degree sequences.

3.3.1 Code

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool isValidSequence(const vector<int>& deg_seq) {
    int sum = accumulate(deg_seq.begin(), deg_seq.end(), 0);
    if (sum % 2 == 1) {
        return false;
    }
    while (!deg_seq.empty()) {
        int d = deg_seq[0];
        if (d > deg_seq.size() - 1) {
            return false;
        }
        deg_seq.erase(deg_seq.begin());
        vector<int> part_1(deg_seq.begin(), deg_seq.begin() + d);
        deg_seq.erase(deg_seq.begin(), deg_seq.begin() + d);
        vector<int> part_2 = deg_seq;
        if (accumulate(part_1.begin(), part_1.end(), 0) % 2 == 0 &&
            accumulate(part_2.begin(), part_2.end(), 0) % 2 == 0) {
            return true;
        }
    }
    return false;
}

int countValidSplits(const vector<int>& deg_seq) {
    vector<int> sorted_seq = deg_seq;
    sort(sorted_seq.rbegin(), sorted_seq.rend());
    int count = 0;
    do {
        if (isValidSequence(sorted_seq)) {
            count++;
        }
    } while (prev_permutation(sorted_seq.begin(), sorted_seq.end()));
    return count;
}

int main() {
    vector<int> degree_sequence = {2,2,3,3,3,4,5,6,6};
    int count = countValidSplits(degree_sequence);
    cout << "Number␣of␣valid␣splits:␣" << count << endl;
    return 0;
}
```

3.3.2 Explanation of code

The isValidSequence function checks whether a degree sequence is a valid split. It takes a vector degseq representing the degree sequence as input. It calculates the sum of all degrees in the sequence using accumulate. If the sum is odd (i.e., sum % 2 == 1), it returns false since a valid split requires an even sum of degrees. The function then enters a loop that continues until the degseq vector is empty. Inside the loop, it checks if the first element d in degseq is larger than the size of the remaining elements in the vector. If it is, the function returns false since it is not possible to split the remaining degrees properly. Otherwise, it creates two vectors, part1, and part2, representing the first d degrees and the remaining degrees, respectively. It checks if both part1 and part2 have an even sum of degrees. If they do, it returns true, indicating a valid split. Otherwise, it continues with the next iteration of the loop, removing the first element of degseq.

The countValidSplits function takes a degree sequence as input and counts the number of valid splits. It first creates a sorted copy of the input degree sequence. It uses sort in reverse order (rbegin and rend) to sort the degrees in descending order. It initializes a counter count to keep track of the number of valid splits. The function then enters a do-while loop. Inside the loop, it checks if the current sorted sequence is a valid split using the isValidSequence function. If it is, it increments the count variable. The loop continues generating the previous permutation of the sorted sequence using prevpermutation until all permutations have been examined. Finally, it returns the count of valid splits.

The main function demonstrates the usage of the code. It initializes a degree sequence in the vector degreesequence. It calls the countValidSplits function with the degree sequence and stores the result in the variable count. It then prints the number of valid splits to the console using cout.

The overall purpose of the code is to determine the number of valid splits that can be obtained from a given degree sequence, where a valid split is defined based on specific criteria implemented in the isValidSequence function.

3.4 Time Complexity

The time complexity is given by:

$$\mathcal{O}\left(\left(\frac{n}{k}+1\right)^{k} \cdot n^{2} \log n \cdot X\right)$$

where n : the number of vertices
k : the number of different degrees
X : the highest degree from all vertices

CHAPTER 4
High Connectivity of Graph

High connectivity of network flow refers to a network or graph that allows for a significant amount of flow to be sent between its vertices while maintaining efficient and unrestricted movement. In other words, it indicates that the network has a robust and flexible capacity for transmitting resources, information, or any other form of flow.

Network flow is typically associated with flow networks, which consist of vertices (nodes) and edges (links) that represent the flow paths. Each edge has a capacity that limits the amount of flow it can carry. The goal is to maximize the flow that can be sent from a source vertex to a sink vertex while respecting the capacity constraints of the edges.

In a highly connected network flow, there are abundant and diverse paths for flow to traverse from the source to the sink. This allows for efficient distribution of resources and facilitates communication between different parts of the network. High connectivity ensures that even if some edges or vertices are disrupted or unavailable, alternative paths can still be used to maintain the flow.

To measure the connectivity of network flow, one common metric is the maximum flow capacity. It represents the maximum amount of flow that can be sent from the source to the sink in the network. A higher maximum flow capacity indicates a more connected network that can handle larger volumes of flow.

In addition to the maximum flow capacity, other measures like minimum cut and edge connectivity can also provide insights into the connectivity of network flow. A smaller minimum cut value or a higher edge connectivity suggests a more interconnected network, enabling the flow to bypass potential bottlenecks or failures.

In summary, a network with high connectivity of network flow possesses abundant flow paths, high maximum flow capacity, and the ability to efficiently distribute resources or information throughout the network, even in the presence of disruptions or limitations.

4.1 Max Flow Min Cut Algorithm

1. In an Undirected graph, the maximum flow from a Source s to a Sink t is equal to the minimum cut separating s and t(max-flow min-cut theorem).
2. At each iteration a path is found from s to t and every edge on that path reduces its capacity by the minimum capacity among all edges of that path. where All edges whose capacity of 0 is deleted.
3. Algorithm termites when there is no path left s-t.[7]

For Example,

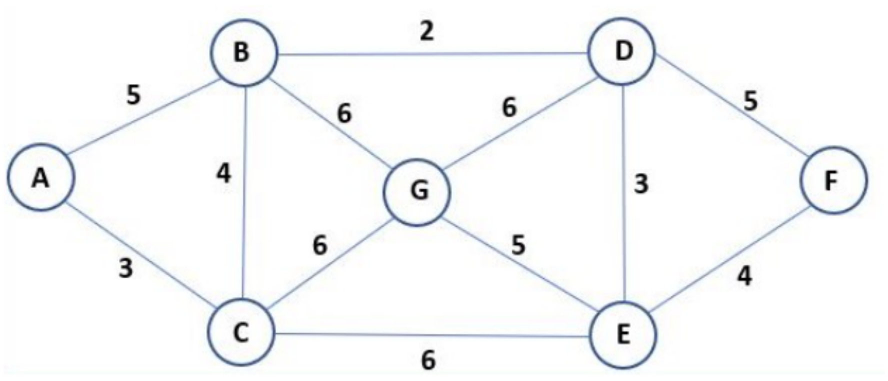


Figure 4.1: Example of Max Flow Min Cut

So, here we consider A as a source and F as a sink

1. First, we go with A->C->E->F with flow 3
2. Then A->B->D->F with flow 2
3. At last A->B->G->D->F with flow 3
4. So the total max flow is 8.
5. For min cut we remove edge AB and AC so we get min cut 8.

4.2 Problem Statement of High Connectivity

Find the Global min-cut and increase it using the 2-switch Method.

4.2.1 Global Min-cut

In a flow network, a global minimum cut represents a partitioning of the vertices into two sets, known as the source-side and sink-side, such that removing the edges between these sets would result in the minimum possible capacity required to disconnect the source from the sink. In other words, a global minimum cut represents the smallest capacity needed to block all paths from the source to the sink in the flow network.]

Here's a step-by-step explanation of finding the global minimum cut in a flow network:

1. Begin with an initial flow in the network. This flow can be determined using various algorithms such as the Ford-Fulkerson algorithm or the Edmonds-Karp algorithm.

2. Use the residual graph derived from the current flow to find an augmenting path from the source to the sink. An augmenting path is a path in the residual graph that allows for additional flow to be sent from the source to the sink.

3. Augment the flow along the augmenting path, increasing the overall flow in the network.

4. Repeat steps 2 and 3 until no more augmenting paths can be found. At this point, the flow is at its maximum capacity.

5. Create a residual graph based on the final flow. In the residual graph, assign capacities to the edges that represent the remaining capacity available for flow. These capacities are determined by subtracting the current flow from the original edge capacities.

6. Perform a graph traversal, such as a depth-first search or a breadth-first search, starting from the source vertex in the residual graph. Mark all reachable vertices as part of the source-side set.

7. The remaining vertices that were not marked during the traversal are part of the sink-side set.

8. The global minimum cut is defined by the set of edges that connect the source-side and the sink-side.

The capacity of the global minimum cut represents the minimum capacity required to disconnect the source from the sink. Removing these edges would block all possible paths from the source to the sink, effectively limiting the flow between them.

Finding the global minimum cut is essential in network analysis as it helps identify critical edges or connections in the flow network that, if disrupted, would significantly affect the overall flow and connectivity between the source and the sink.

4.3 Maximize the Global min-cut using 2-switch

1. Now, first find the global min-cut of the graph using the max flow min cut algorithm.
2. choose two edges that were not cross the global min-cut suppose (x,y) and (v,w). Remove the edges (x,y) and (v,w) from the graph.
3. Add the edges (x,v) and (y,w) to the graph.
4. Recalculate the min-cut of the new graph using the max flow min cut algorithm

If the new min-cut is greater than the previous min-cut, then the 2-switch has increased the global min-cut of the graph.

If the new min-cut is less than or equal to the previous min-cut, then the 2-switch has not increased the global min-cut and should be undone by reversing steps 3 and 4.

You can repeat the 2-switch procedure by choosing different pairs of edges that were not cross the current global min-cut until no further increase in the global min-cut is possible.

4.3.1 code

```
    #include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

// Structure to represent an edge in the graph
struct Edge {
    int source, destination, capacity, flow;
    Edge(int source, int destination, int capacity) : source(source)
    , destination(destination), capacity(capacity), flow(0) {}
};

// Class for the graph
class Graph {
    int numVertices;
    vector<Edge> edges;
    vector<vector<int>> adjacencyMatrix;

public:
    Graph(int numVertices) : numVertices(numVertices) {
        adjacencyMatrix.resize(numVertices,
        vector<int>(numVertices, 0));
    }

    void addEdge(int source, int destination, int capacity) {
        Edge edge(source, destination, capacity);
        edges.push_back(edge);
        adjacencyMatrix[source][destination] = edges.size() - 1;
        // Index of the edge in the edges vector
    }

    bool bfs(vector<vector<int>>& residualGraph, vector<int>& parent
    , int source, int sink) {
        vector<bool> visited(numVertices, false);
        queue<int> q;

        q.push(source);
        visited[source] = true;
        parent[source] = -1;

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            for (int v = 0; v < numVertices; v++) {
                if (!visited[v] && residualGraph[u][v] > 0) {
                    q.push(v);
                    parent[v] = u;
                    visited[v] = true;
                }
            }
        }

        return visited[sink];
    }

    int fordFulkerson(int source, int sink) {
        vector<vector<int>> residualGraph(adjacencyMatrix);

        int maxFlow = 0;
        vector<int> parent(numVertices);

        while (bfs(residualGraph, parent, source, sink)) {
            int pathFlow = numeric_limits<int>::max();

            // Find the minimum residual capacity in the path
            for (int v = sink; v != source; v = parent[v]) {
                int u = parent[v];
                pathFlow = min(pathFlow, residualGraph[u][v]);
            }

            // Update the residual capacities
            and reverse edges along the path
            for (int v = sink; v != source; v = parent[v]) {
                int u = parent[v];
                residualGraph[u][v] -= pathFlow;
                residualGraph[v][u] += pathFlow;
            }

            maxFlow += pathFlow;
        }
        return maxFlow;
    }
};

// Function to find the global minimum cut
using max flow min cut algorithm
void findGlobalMinCut(Graph& graph, int source, int sink) {
    // Step 1: Find the max flow using Ford-Fulkerson algorithm
    int maxFlow = graph.fordFulkerson(source, sink);

    // Step 2: Iterate over all edges and
    find those crossing the minimum cut
    for (const auto& edge : graph.edges) {
        int u = edge.source;
        int v = edge.destination;
        int capacity = edge.capacity;

        // If the edge is from the source side and
        leads to the sink side in the residual graph
        if (graph.adjacencyMatrix[u][v] != -1
        && graph.adjacencyMatrix[v][u] == -1) {
            // If the edge is saturated in the original graph
            if (edge.flow == capacity) {
                cout << "Edge␣(" << u << ",␣" << v << ")
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣crosses␣the␣global␣minimum␣cut." << endl;
            }
        }
    }

    // Step 3: Update the graph by removing and adding edges
    for (auto& edge : graph.edges) {
        int u = edge.source;
        int v = edge.destination;

        // If the edge crosses the global minimum cut,
        remove it and add the reversed edge
        if (graph.adjacencyMatrix[u][v] != -1
            && graph.adjacencyMatrix[v][u] == -1
            && edge.flow == capacity) {
                graph.addEdge(u, v, edge.capacity);
                graph.addEdge(v, u, 0);
            }
        }

    // Step 4: Recalculate the minimum cut of the updated graph
    int newMinCut = graph.fordFulkerson(source, sink);

    cout << "Updated␣minimum␣cut:␣" << newMinCut << endl;
}

int main() {
    int numVertices = 4;
    Graph graph(numVertices);

    // Add edges to the graph
    graph.addEdge(0, 1, 3);
    graph.addEdge(0, 2, 2);
    graph.addEdge(1, 2, 1);
    graph.addEdge(1, 3, 1);
    graph.addEdge(2, 3, 3);

    int source = 0;
    int sink = 3;

    // Find the global minimum cut and update the graph
    findGlobalMinCut(graph, source, sink);

    return 0;
}
```

4.3.2 Explanation of code

This code begins with including necessary libraries and the definition of structures and classes. The Edge structure represents an edge in the graph, containing information such as source, destination, capacity, and flow. The Graph class represents the graph itself and provides methods to add edges, perform a breadth-first search (BFS), and find the maximum flow using the Ford-Fulkerson algorithm.

The Graph class maintains an adjacency matrix to store the graph connections and a vector of edges to keep track of the edges added. The addEdge function adds an edge to the graph by updating the adjacency matrix and the edges vector.

The bfs function performs a BFS on the residual graph (which is initially the same as the adjacency matrix) to find an augmenting path from the source to the sink. It uses a queue and a visited array to keep track of visited vertices and a parent array to store the parent vertex of each visited vertex.

The fordFulkerson function implements the Ford-Fulkerson algorithm. It repeatedly calls the bfs function to find augmenting paths in the residual graph and updates the flow along those paths. It returns the maximum flow obtained.

The findGlobalMinCut function uses the max flow obtained from the Ford-Fulkerson algorithm to identify the global minimum cut. It iterates over all edges in the graph and checks if an edge crosses the minimum cut by verifying if it is saturated in the original graph and not in the residual graph. It then updates the graph by removing the crossing edges and adding their reversed counterparts. Finally, it recalculates the minimum cut of the updated graph.

In the main function, a sample graph is created, edges are added, and the findGlobalMinCut function is called to find the global minimum cut and update the graph accordingly.

Overall, the code demonstrates the process of finding the maximum flow and identifying the global minimum cut in a graph using the Ford-Fulkerson algorithm.

4.4 Find Number of 2-switch

1. Initialize the number of 2 switches to zero. Repeat the following steps until no more than 2 switches can be performed: Compute the min-cut of the current graph.
2. For each pair of edges (u,v) and (x,y) that are not adjacent, compute the min-cut of the graph by removing the edges (u,v) and (x,y) and adding edges of (u,x) and (v,y) or any other perfect matching which is not present.
3. After performing the graph have a smaller min cut than the previous graph, then increment the number of 2 switches.
4. Return the number of 2 switches.

4.4.1 code

```
    #include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

// Structure to represent an edge in the graph
struct Edge {
    int source, destination, capacity, flow;

    Edge(int source, int destination, int capacity) : source(source),

    destination(destination), capacity(capacity), flow(0) {}
};

// Class for the graph
class Graph {
    int numVertices;
    vector<Edge> edges;
    vector<vector<int>> adjacencyMatrix;
```

```
public:
    Graph(int numVertices) : numVertices(numVertices) {
        adjacencyMatrix.resize(numVertices,
        vector<int>(numVertices, 0));
    }

    void addEdge(int source, int destination, int capacity) {
        Edge edge(source, destination, capacity);
        edges.push_back(edge);
        adjacencyMatrix[source][destination] = edges.size() - 1;
        // Index of the edge in the edges vector
    }

    bool bfs(vector<vector<int>>& residualGraph, vector<int>& parent
    , int source, int sink) {
        vector<bool> visited(numVertices, false);
        queue<int> q;

        q.push(source);
        visited[source] = true;
        parent[source] = -1;

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            for (int v = 0; v < numVertices; v++) {
                if (!visited[v] && residualGraph[u][v] > 0) {
                    q.push(v);
                    parent[v] = u;
                    visited[v] = true;
                }
            }
        }

        return visited[sink];
    }
```

```
int fordFulkerson(int source, int sink) {
    vector<vector<int>> residualGraph(adjacencyMatrix);

    int maxFlow = 0;
    vector<int> parent(numVertices);

    while (bfs(residualGraph, parent, source, sink)) {
        int pathFlow = numeric_limits<int>::max();

        // Find the minimum residual capacity in the path
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            pathFlow = min(pathFlow, residualGraph[u][v]);
        }

        // Update the residual capacities
        and reverse edges along the path
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            residualGraph[u][v] -= pathFlow;
            residualGraph[v][u] += pathFlow;
        }

        maxFlow += pathFlow;
    }

    return maxFlow;
}

int computeMinCut(int source, int sink) {
    return fordFulkerson(source, sink);
}

void performTwoSwitches() {
    int numTwoSwitches = 0;
    bool twoSwitchesPerformed = true;
```

```
while (twoSwitchesPerformed) {
    twoSwitchesPerformed = false;
    int previousMinCut = computeMinCut(0, numVertices - 1);

    for (int u = 0; u < numVertices; u++) {
        for (int v = u + 1; v < numVertices; v++) {
            for (int x = 0; x < numVertices; x++) {
                for (int y = x + 1; y < numVertices; y++) {
                    // Check if edges (u,v) and
                        (x,y) are not adjacent
                    if (adjacencyMatrix[u][v] == -1
                        && adjacencyMatrix[x][y] == -1) {
                        // Remove edges (u,v) and (x,y)
                        adjacencyMatrix[u][v] =
                        adjacencyMatrix[v][u] = -1;
                        adjacencyMatrix[x][y] =
                        adjacencyMatrix[y][x] = -1;

                        // Add edges (u,x) and (v,y)
                        or any other perfect matching
                        adjacencyMatrix[u][x] =
                        adjacencyMatrix[x][u] = edges.size();
                        edges.push_back(Edge(u, x, 1));
                        adjacencyMatrix[v][y] =
                        adjacencyMatrix[y][v] = edges.size();
                        edges.push_back(Edge(v, y, 1));

                        // Compute the min cut of
                        the updated graph
                        int newMinCut =
                        computeMinCut(0, numVertices - 1);

                        // If the new min cut is smaller
                        than the previous min cut,
                        increment the number of 2 switches
                        if (newMinCut < previousMinCut) {
```

```
                                    numTwoSwitches++;
                                    twoSwitchesPerformed = true;
                                    previousMinCut = newMinCut;
                                }
                                else {
                                    // Revert the changes
                                    adjacencyMatrix[u][x] =
                                    adjacencyMatrix[x][u] = -1;
                                    adjacencyMatrix[v][y] =
                                    adjacencyMatrix[y][v] = -1;
                                    adjacencyMatrix[u][v] =
                                    adjacencyMatrix[v][u] =
                                    edges.size();
                                    edges.push_back(Edge(u, v, 1));
                                    adjacencyMatrix[x][y] =
                                    adjacencyMatrix[y][x] =
                                    edges.size();
                                    edges.push_back(Edge(x, y, 1));
                                }
                            }
                        }
                    }
                }
            }
        }

        cout << "Number␣of␣2␣switches:␣" << numTwoSwitches << endl;
    }
};

int main() {
    int numVertices = 5;
    Graph graph(numVertices);

    // Add edges to the graph
    graph.addEdge(0, 1, 1);
    graph.addEdge(0, 2, 1);
    graph.addEdge(1, 2, 1);
    graph.addEdge(1, 3, 1);
    graph.addEdge(2, 4, 1);
    graph.addEdge(3, 4, 1);

    // Perform the two switches and get the number of 2 switches
    graph.performTwoSwitches();

    return 0;
}
```

4.4.2 Explanation of code

This code implements the Two-Switches algorithm for finding the minimum number of edge-disjoint paths in a graph. The code starts with including necessary libraries and declaring the required structures and variables.

The Edge structure represents an edge in the graph and stores the source, destination, capacity, and flow of the edge.

The Graph class is defined to represent the graph and contains functions for adding edges, performing the Ford-Fulkerson algorithm to find the maximum flow, and performing the Two-Switches algorithm.

The Graph constructor initializes the adjacency matrix and takes the number of vertices as a parameter.

The addEdge function adds an edge to the graph by creating an Edge object and updating the adjacency matrix.

The bfs function performs a breadth-first search on the residual graph to find an augmenting path from the source to the sink. It uses a queue and a visited array to keep track of visited vertices.

The fordFulkerson function implements the Ford-Fulkerson algorithm to find the maximum flow in the graph. It repeatedly calls the bfs function to find augmenting paths and updates the residual graph and flow along the path until no more augmenting paths exist. It returns the maximum flow.

The computeMinCut function calculates the minimum cut in the graph by calling the fordFulkerson function and passing the source and sink vertices.

The performTwoSwitches function implements the Two-Switches algorithm. It iterates through all possible combinations of edges (u, v) and (x, y) where u, v, x, and y are vertices. It checks if the edges (u, v) and (x, y) are not adjacent in the graph and performs the switch operation by removing edges (u, v) and (x, y) and adding edges (u, x) and (v, y). It then computes the new minimum cut and checks if it is smaller than the previous minimum cut. If so, it increments the number of two switches performed and continues to the next iteration. Otherwise, it reverts the changes.

In the main function, a Graph object is created, edges are added to the graph, and the performTwoSwitches function is called to find the minimum number of two switches needed.

Finally, the code outputs the number of two switches performed.

CHAPTER 5
Conclusion

In this thesis, we have explored the inferences that can be made on the possible number of components of a graph, on the basis of its degree sequence alone. This is done by devising algorithms that obtain all possible ways to split a given degree sequence into two parts and applying recursion on the fragments to further split them. We have also explored the maximum possibility of a graph given only the degree sequence. This was done using the concepts of 2 switches together with standard max flow min cut algorithms.

This work constitutes a part of the broader research problem of inferences that can be made about graphs on the basis of just their degree sequence. This problem is of interest since the degree sequence of a graph is one of its important attributes, but at the same time, several pairwise non-isomorphic graphs often share the same degree sequence. This work should help in a complete characterization of realization graphs, as defined in this thesis.

Work on characterizing realization graphs completely, will either directly or indirectly build on the work done here, among other things. The results here will lead to some immediate fallouts like some forced and/or forbidden structures in realization graphs

References

[1] M. D. Barrus. On 2-switches and isomorphism classes. *Discrete Mathematics*, 312(15):2217–2222, 2012.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. *Journal of the Operational Research Society*, 42, 2001.

[3] S. L. Devadoss. A realization of graph associahedra. *Discrete Mathematics*, 309(1):271–276, 2009.

[4] J. R. Griggs and D. J. Kleitman. Independence and the havel-hakimi residue. *Discrete Mathematics*, 127(1-3):209–212, 1994.

[5] S. L. Hakimi. On realizability of a set of integers as degrees of the vertices of a linear graph. i. *Journal of the Society for Industrial and Applied Mathematics*, 10(3):496–506, 1962.

[6] V. Havel. A remark on the existence of finite graph (hungarian). *Casopis Pest., Mat.*, 80:477–480, 1955.

[7] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, 46(6):787–832, 1999.

[8] S.-Y. R. Li. Graphic sequences with unique realization. *Journal of Combinatorial Theory, Series B*, 19(1):42–68, 1975.

[9] R. Milo, N. Kashtan, S. Itzkovitz, M. E. Newman, and U. Alon. On the uniform generation of random graphs with prescribed degree sequences. *arXiv preprint cond-mat/0312028*, 2003.

[10] D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.