University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations | Theses, Dissertations, and Major Papers

Fall 2021

Generating graphs from degree sequence

Amreeth Rajan Nagarajan
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Nagarajan, Amreeth Rajan, "Generating graphs from degree sequence" (2021). *Electronic Theses and Dissertations*. 8864.
https://scholar.uwindsor.ca/etd/8864

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

Generating graphs from degree sequences

By

Amreeth Rajan Nagarajan

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor
Windsor, Ontario, Canada
2021

© 2021 Amreeth Rajan Nagarajan

Generating graphs from degree sequences

by

Amreeth Rajan Nagarajan

APPROVED BY:

Y. Aneja
Odette School of Business

D. Wu
School of Computer Science

A. Mukhopadhyay, Advisor
School of Computer Science

August 26, 2021

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

A graph consists of vertices and edges, connecting pairs of vertices. The subject of graph generation has a long history. Initial research focused on creating catalogues of graphs of small size. It has proved useful for creating input instances for different graph algorithms as well as for disproving conjectures or formulating new ones for various graph classes.

A problem that has garnered a lot of attention is that of deciding if a given sequence of n integers $d_1 \geq d_2 \geq d_3 \ldots d_n$ is graphical, that is, does there exist a simple graph whose degrees are the d_i's. Necessary and sufficient conditions for this have been obtained, among which the most notable ones are those of Erdos & Gallai and Havel & Hakimi. In our work, we have studied the extended problem of constructing a forest of trees from such a degree sequence as well split graphs (a split graph is a chordal graph whose complement is also chordal).

Given a degree sequence, there can be many graphs that satisfy it. An important problem is to be able to sample uniformly at random sample from the space of all possible graphs that satisfy a given degree sequence. This has been extensively studied by researchers in the area of complex networks (for example social, neural, metabolic, protein-protein interaction networks).

In our work we have studied both the sampling problem as well as the enumeration problem. We have improved upon previously established enumeration methods and have turned them into sampling methods. The different methods have been compared and their complexities have been analyzed.

DEDICATION

To all the people in my life who've supported me and had my back no matter what.

ACKNOWLEDGEMENTS

I would firstly like to express my earnest gratitude towards my supervisor, Dr. Asish Mukhopadhyay for his invaluable feedback, suggestions, regular interactions and patience, which has taught me a great deal and has helped me write this thesis. His ideas and teachings have always inspired me and has helped me break my own limits with learning. I would like to thank my thesis committee Dr.Dan Wu, and Dr. Yash Aneja for taking the time out of their busy schedules to make this possible.

I am extremely grateful to the Computer Science faculty members, Graduate Secretary and the other important helping staff behind the scenes who have extended their help at every point of my thesis journey. I would like to extend my gratitude to my friends Anjali, Chinnu, Monica, Pooja, Rashi, and Rida for offering me their time and support at every step of the way.

I would finally like to thank my parents, brother, sister-in-law, and other family members for being there by my side and giving me all the reassurance I needed. This journey wouldn't have been possible without their persistent encouragement and love.

TABLE OF CONTENTS

DECLARATION OF ORIGINALITY **iii**

ABSTRACT **iv**

DEDICATION **v**

ACKNOWLEDGEMENTS **vi**

LIST OF FIGURES **ix**

LIST OF TABLES **xi**

1 Introduction **1**

1.1 Preliminaries 1

1.2 Graph Generation 2

1.2.1 Problem statement 3

1.3 The motivation behind the problem 3

1.4 Thesis organization 4

1.5 Conditions for checking graphicality 4

1.5.1 Havel-Hakimi condition 5

1.5.2 Erdos-Gallai condition 5

2 Construction algorithms to generate graphs from degree sequences **6**

2.1 Hakimi's graph construction 6

2.2 Tripathi's construction based on Erdos-Gallai 9

3 Special cases **17**

3.1 Special case: Forests 17

3.2 Special case 2: Split Graphs 20

3.2.1 Applications of split graph 20

3.2.2 The Recognition algorithm 22

4 Exhaustive methods **25**

4.1 James Riha exhaustive algorithms 25

4.2 Kim's exhaustive algorithm 30

4.3 Comparing the exhaustive algorithms: James Riha exhaustive algorithm VS Kim-1 exhaustive algorithm 34

4.3.1 Number of graphs generated 34

4.3.2 Isomorphic repetitions 35

4.3.3 Rejections 36

5 Sampling methods and comparative studies **38**

5.1 Modification of the exhaustive methods into sampling methods 38

5.2 Application of sampling algorithms 39

5.3 Modified James-Riha sampling algorithm 40

5.4 Modified Kim's sampling method 43

5.5 Kim's follow-up sampling algorithm 46

5.6 Complexity Analysis 51

5.6.1 James Riha sampling method 51

5.6.2 Modified Kim-1 sampling method 51

5.6.3 Kim's second sampling method 52

5.7 James Riha exhaustive VS James Riha sampling 53

5.8 Kim exhaustive VS Kim modified sampling 54

5.9 Comparing the modified methods: James Riha sampling vs. Kim 1 sampling 55

6 Concluding Thoughts **57**

6.1 Future Works 58

Bibliography **59**

Vita Auctoris **61**

LIST OF FIGURES

1.1 Example graph 1

1.2 Graph with degree sequence (2,2,2,2) 2

2.1 Graph generated by Hakimi's algorithm; N: Node labels, D: Degree sequence, D_r: Residual degree sequence after saturating hub node . . . 7

2.2 Tripathi case 0 9

2.3 Tripathi case 1.1; r is deficient by 2 10

2.4 Tripathi case 1.2; r is deficient by 1 10

2.5 Tripathi case 2 11

2.6 Tripathi case 3 11

2.7 Steps in generating a graph of degree sequence [3,2,2,1,1,1] using Tripathi's method; (i) r=1; (ii) r=2; (iii) r=3 12

3.1 Steps in generating a forest realizing [3,2,2,1,1,1,1,1] 18

3.2 Split graph example 20

3.3 Split graph social network detection 20

3.4 Split graph characteristics: Split graph G and Split graph $\overline{G}$ 21

3.5 Split graph characteristics: Multiple split graphs with degree sequence [4,4,3,3,1,1] 21

3.6 Generation of clique 22

3.7 Connecting nodes of the independent set to the clique 23

4.1 Steps in generating graphs using James Riha algorithm; 26

4.2 Graph 1 generated by exhaustive James Riha algorithm 27

4.3 Graph 2 generated by exhaustive James Riha algorithm 27

4.4 Example of one of the graphs generated by the Kim exhaustive algorithm 31

4.5 The collection of graphs generated by exhaustive Kim algorithm 32

4.6 Graph generated by James Riha exhaustive algorithm for degree sequence (3,3,2,2,2,2) 34

4.7 Graph missed out by James Riha algorithm 34
4.8 Graph generated by James-Riha exhaustive algorithm for degree sequence (2,2,2,1,1) 35
4.9 Graph generated by Kim's exhaustive algorithm 35
4.10 Rejections faced by the James-Riha exhaustive method 36
4.11 Rejection free Kim's exhaustive method 37

5.1 The different paths traversed in the James Riha sampling algorithm . . 41
5.2 The graphs generated James Riha modified algorithm to make it sample 42
5.3 Graph generated by the modified Kim algorithm 44
5.4 Kim's sampling algorithm: Saturating node 1 47
5.5 Kim's sampling algorithm: Saturating node 2 48
5.6 Kim's sampling algorithm: Saturating node 3 48
5.7 Side by side comparison of James Riha exhaustive method (i)James Riha exhaustive (ii)James Riha sampling advantageous run (iii)James Riha sampling wasteful run 53
5.8 Side by side comparison of Kim's first method and its modification (i)Kim exhaustive generation (ii)Modification of Kim's method for sampling . . 54
5.9 Graph comparing the run times of James Riha sampling and Kim 1 sampling 55

LIST OF TABLES

5.1 Recording of number of failures before a graph is generated for 10 cases (10 trials for each case) 54

5.2 A tabulation comparing the run times of JR-Ran and Kim 1-Ran for 10 different cases (10 trials each) 56

Chapter 1

Introduction

Graphs have been studied for years and years due to their versatile nature. Graphs can be seen anywhere, starting from a simple social network down to a protein-protein interaction network. With its ability to represent a variety of problems, it has applications in multiple fields resulting in studies on related problems such as generation, enumeration, induced sub-graphs, graph coloring, decomposition, graph classes, etc.

1.1 Preliminaries

A graph is a collection of nodes and edges and can be represented as $G = (V, E)$, where V denotes the set of vertices or nodes, and E denotes the set of edges connecting a pair of vertices. Figure 1.1 shows a graph with 5 nodes and 6 edges.

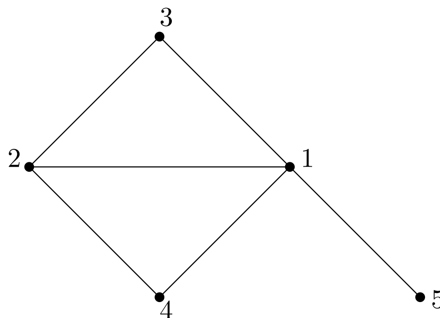


FIGURE 1.1: Example graph

Two nodes $u, v \in V$, are adjacent if they have an edge joining them, i.e., $(u,v) \in E$. For example, nodes 3 and 4 are adjacent in Figure 1.1, but 1 and 4 are non-adjacent. The neighbourhood $N(v)$ of a vertex v, is the subset of vertices such that $u \in V|(u, v) \in E$.

The degree of a vertex v is the number of edges incident on it, i.e., $|N(v)|$. For a graph on n vertices labeled from 1 to n, the degree sequence is of the form $d_1 \geq d_2 \geq \ldots \geq d_n$, and can be defined as a non-increasing sequence of its vertex degrees. For example, the degree sequence of the graph in Figure 1.1 would be [4,3,2,2,1].

A chord can be defined as an edge that isn't part of a cycle but joins a pair of non-adjacent and non-consecutive vertices. In Figure 1.1, edge (1,2) can be considered the chord in the cycle 1-3-2-4. A graph G can be considered a chordal graph if and only if all cycles of four or more vertices have a chord. The graph in Figure 1.1 is an example of such graphs. A clique can be characterized as the subset of vertices in an undirected graph, where every two distinct vertices are adjacent, i.e., its induced subgraph is complete.

There are many different types of graphs. They can be labeled or unlabeled, directed, or undirected. The graphs considered in this paper are labeled and undirected. The graphs will neither have self-loops nor have parallel edges.

1.2 Graph Generation

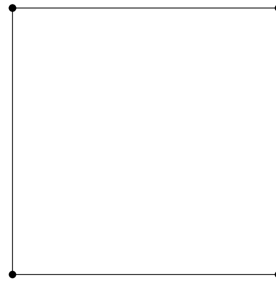


FIGURE 1.2: Graph with degree sequence (2,2,2,2)

A sequence of positive integers can be considered graphical if and only if a graph has vertices realizing these degrees. The sequence of integers, i.e. $[2, 1, 1, 1]$, cannot be considered a degree sequence since no graph has vertices that realize these degrees. Whereas the sequence of integers, i.e. $[2, 2, 2, 2]$, can be classed as a degree sequence since a graph, as seen in Figure 1.2, realizes the degrees in the sequence.

1.2.1 Problem statement

This thesis aims to address the problems that are associated with generating graphs. Graph generation can be seen as the process of constructing graphs of a certain class.

There are two types of generation addressed in this thesis: Exhaustive generation and Random Generation / Sampling. Exhaustive generation suffers in terms of complexity when it comes to realistic applications of graph generation. A solution to this is to sample a single graph at random from a space of all possible graphs representing the given degree sequence. In this thesis, we will be looking at some of the ways this can be achieved and the difficulties faced while trying to do so.

1.3 The motivation behind the problem

Graph generation has applications in several fields. The following are some of its theoretical applications.

1. Creating catalogues of graphs to create stores of graphs to sample from, serves to aid studies for graph theorists [1]. Conjectures are often made about classes of graphs, and this can be done with the help of the example graphs generated. For instance, the conjecture where all regular graphs are said to have Hamiltonian cycles can be established by generating examples of the class of graphs, i.e., regular graphs in this case, and then observing them.
2. Similarly conjectures about a class of graphs can be refuted by generating counterexamples of the same to contradict it.
3. Algorithms surrounding specific classes of graphs often require test cases provided by graph generation methods. For example, an algorithm that tests if a graph is strongly chordal or not, would require a graph to be generated in the first place for testing.

In particular, sampling has been seen to have several real-world applications. These applications will be covered in the later sections.

1.4 Thesis organization

The thesis has four chapters, and the following is the description of each of them.

Chapter 1: The basic concepts and defining terms are established in this chapter. This includes descriptions for graphs, graph generation, and graphicality.

Chapter 2: This chapter consists of the conditions needed to check if a sequence is graphical. Some of these conditions are the Erdos-Gallai condition, Havel-Hakimi condition. Further, two constructive methods are discussed: the Tripathi method [2] that uses the Erdos-Gallai condition, and the Hakimi method [3] that uses the Havel-Hakimi condition.

Chapter 3: Chapter 3 is a discussion on two special cases, namely Forests and Split graphs. These methods are extensions of graph generation principles to specific subtypes.

Chapter 4: Chapter 4. begins with a brief discussion on two exhaustive methods by James Riha [4] and Kim [5]. The methods are discussed in detail, then followed by a description of how these exhaustive generation methods can be modified to achieve sampling.

Chapter 5: Chapter 5 discusses the modified James Riha method and a modified Kim's method. There is also a discussion on a follow-up method by Kim [6] that samples instead of generating graphs exhaustively. These methods are compared and contrasted based on several factors. The complexities of these methods are also analyzed.

Chapter 4: This chapter includes concluding remarks with future directions that can be taken with the research.

1.5 Conditions for checking graphicality

The following are two well-known necessary and sufficient conditions for a sequence of non-negative integers to be graphical.

1.5.1 Havel-Hakimi condition

A non-increasing sequence of non-negative integers d_1, d_2, ..., d_n can be proven to be graphical if and only if the derived sequence $d_2 - 1$, $d_3 - 1$, ..., $d_{d_1+1} - 1$, d_{d_1+2}, d_{d_1+3}, ..., d_n can also be proven to be graphical. The derived sequence is obtained by joining the first node to nodes in the increasing order of their degrees till d_1 is reduced to zero, which reduces the degree of the other nodes by 1.

This condition was first published by Havel[7] in 1955 and later rediscovered by Hakimi in 1962, which is why the result is attributed to both Havel and Hakimi [8].

1.5.2 Erdos-Gallai condition

The Erdos-Gallai[9] condition states that a non-increasing sequence of non-negative integers d_1 , d_2, ..., d_n can be realized as a graph if and only if

1. $\sum_{i=1}^{n} d_i$ is even
2. $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i)$ for $1 \leq k \leq n$

The next section contains two constructive algorithms based on these conditions, where a graph can be built from a given degree sequence.

Chapter 2

Construction algorithms to generate graphs from degree sequences

2.1 Hakimi's graph construction

The following construction method is based on the Havel-Hakimi [3] condition stated in the previous section. The input to the algorithm is n i.e. the number of nodes, and the degree sequences i.e. a non-increasing sequence of integers d_1, d_2, $\ldots$, d_n. The nodes are labeled 1, 2, $\ldots$, n such that $d_1 \geq d_2 \geq \ldots \geq d_n$.

The algorithm originally connects the largest node to the other available nodes in non-increasing order of their residual degree at each step. This process is repeated until all the nodes are fully saturated. A modification of the method can be described as follows.

Let k be the hub node that is randomly selected to be saturated. There are two variables namely $leftTorightIndex$ and $rightIndex$. $leftTorightIndex$ corresponds to the node selected to be connected to node k, and $rightIndex$ restricts the movement of the $leftTorightIndex$. At each step, $leftTorightIndex$ ranges from the $k+1^{th}$ node and gets incremented until node k is saturated and does not go beyond $rightIndex$.

The modified method checks for graphicality simultaneously while building the graph and makes appropriate rejections when a sequence is detected as non-graphical at any

point of the construction.

Consider the construction of a graph on 5 nodes with degrees [2,2,2,1,1]. A 2D array *Seq* is initialized to maintain node label-degree integrity. *Seq* for the example would initially be [[1,2],[2,2],[3,2],[4,1],[5,1]].

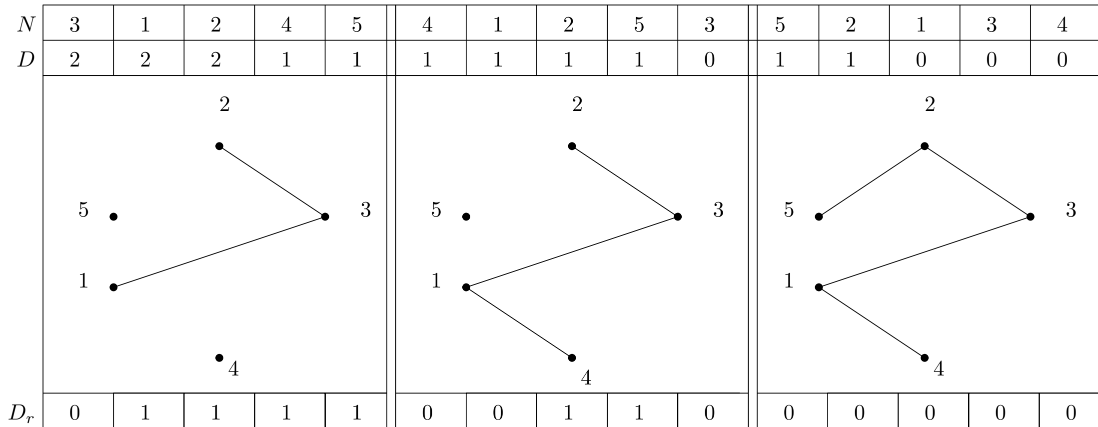


FIGURE 2.1: Graph generated by Hakimi's algorithm; N: Node labels, D: Degree sequence, D_r: Residual degree sequence after saturating hub node

Node 3 is selected as the hub node (i.e., the node to saturate) and is moved to the beginning of the list. Edges (3,1) and (3,2) are introduced thereby resulting in a residual sequence of [1,1,1,1]. This leads to node 3 being saturated completely and being pushed to the end of *Seq*.

Instead of sorting the degree sequence (like in the original method), the sorted degree sequence required for the successive step is obtained through a simple merge process. This merge is not computationally expensive since the degrees of nodes used to saturate the hub node remain sorted, and so do the degrees of nodes not used to saturate. These two sorted lists can be merged to obtain the sorted list for the next step. In the above example, after saturating node 3, you would merge [[1,1],[2,1]] with [[4,1],[5,1]] which would result in [[1,1],[2,1],[4,1],[5,1]] for the successive step. The same process is repeated until all the nodes are saturated, and the residual degree sequence is all zeroes.

The algorithm can be formally stated as follows.

Algorithm 1: HakimiGen(parameters: n, Seq)

Data: n: Number of nodes, Seq: 2D array [[Node label, Degree],. . .]

Result: $G : (V, E)$ Graph realizing degrees in Seq,$IsGraphical$: State of graphicality

```
1  OriginalSeq ← Degrees in Seq;
2  rightIndex ← n − 1;
3  while rightIndex ≥ Seq[0][1] AND Seq[0][1] > 0 do
4  |   k ← random node from Seq ;
5  |   put k at the start of Seq;
6  |   leftTorightIndex ← 1;
7  |   while k not saturated] do
8  |   |   Add (leftTorightIndex, k) to E;
9  |   |   update Seq and increment leftTorightIndex;
10 |   end
11 |   Seq ← Merge Seq[1 : leftTorightIndex − 1] and
   |     Seq[leftTorightIndex : rightIndex];
12 |   remove k from Seq and append it to the end;
13 |   move rightIndex to the left of the first non-zero element from Seq;
14 end
15 if OriginalSeq[0] > rightIndex then
16 |   IsGraphical ←False
17 end
18 else
19 |   IsGraphical ←True
20 return G,IsGraphical
```

Complexity analysis

The complexity of the algorithm can be computed as follows. In the worst case, the time required to add the edges, i.e., O($|E|$), is O(n^2).

This is because the amortized complexity of the merges over all the steps is O(n^2). Thus the complexity is O($n^2 + |E|$) which is O(n^2).

2.2 Tripathi's construction based on Erdos-Gallai

Tripathi's method [2] iterates over each node and constructs the final graph by constructing intermediate sub realizations. At each node, the residual degree sequence is analyzed to see how the construction proceeds. Before iterating through to subsequent nodes, the subgraph generated is checked for graphicality using the modified Erdos-Gallai condition. r indicates the node that's picked to saturate. A total of five possible cases exist when deciding how a chosen node r is saturated. The method iterates through these five cases to generate sub-realizations to build a graph that satisfies a given degree sequence.

The array in Figures below depicts the degree sequence. The left subset stands for the indices that have already been saturated, and the right subset is those that haven't yet been iterated through.

Case 0

Case 0 is the straightforward case wherein we still have nodes in the right subset to connect r to (as seen in Figure 2.2).

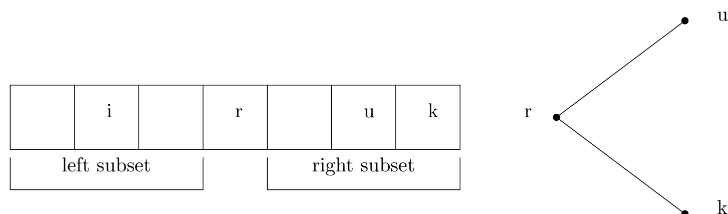


Figure 2.2: Tripathi case 0

Case 1

Case 1 is entered when no more nodes are left in the right subset to connect r to. There are nodes in the left subset that r hasn't been connected to, but they're fully saturated. Case 1 has two sub-cases.

(1.1) The first sub case can be seen in Figure 2.3 where r is deficient by 2, two nodes i from the left subset, and u from the entire set such that $(i, u) \in E$. Node r is introduced between u and i, by replacing edge (u, i) with edges (u, r) and (i, r) thereby increasing r's degree by 2.

(1.2) The second sub case as seen in Figure 2.4. This case occurs when we find that r has already been connected to a node k from the right subset, but is still deficient by 1. We find a node i from the left subset, and a node u from the entire set (excluding r)

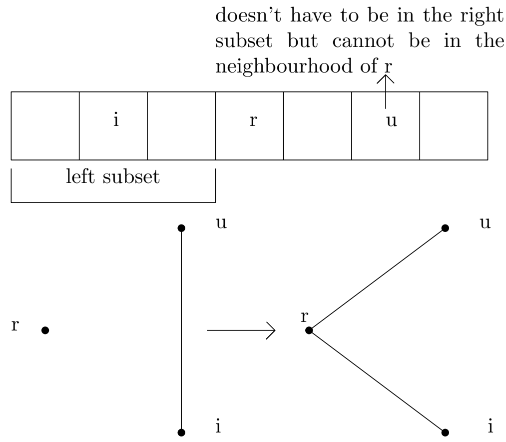


FIGURE 2.3: Tripathi case 1.1; r is deficient by 2

such that $(r, u) \notin E$ and $(u, i) \notin E$. The edge (r, k) is replaced with edges (r, u) and (r, i).

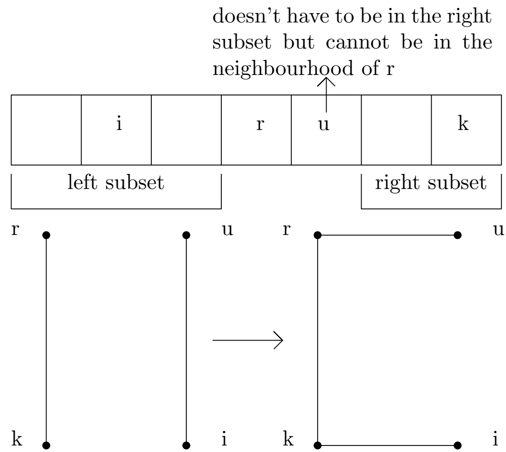


FIGURE 2.4: Tripathi case 1.2; r is deficient by 1

Case 2

Case 2 is when r has been connected to all the nodes in the left subset, and no nodes from the right subset can be connected to r. Case 0 would apply if the right subset consisted of nodes that weren't already connected to r (as seen in Figure 2.5).

We find a node k from the right subset, a node i from the left subset, and a node u from the entire set (excluding r) such that $(k, r), (u, i) \in E$. We replace edge (u, i) with edges (r, u) and (k, i).

Case 3

In case 3, all the nodes in the left subset have been connected to r, as shown in Figure 2.6. We find two nodes i and j from the left subset such that $i < j$, and two nodes u and w from the entire set (excluding r) such that $(i, u), and(j, w) \in E$ and $(u, r) and(w, r) \notin E$.

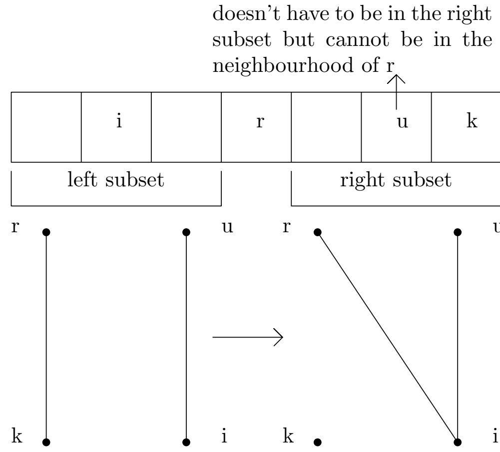


FIGURE 2.5: Tripathi case 2

u and w can be the same. We replace the edges (i, u) and (j, w) with edges (i, j) and (r, u).

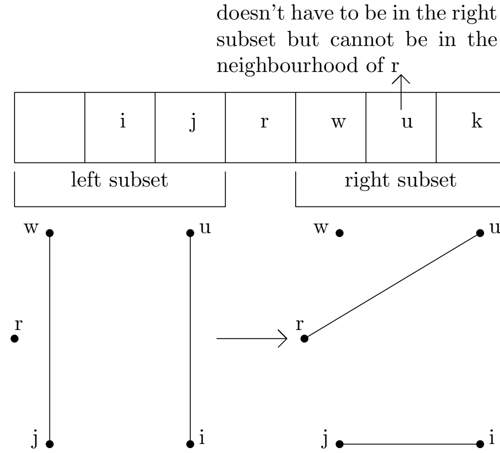


FIGURE 2.6: Tripathi case 3

Case 4

The last case is entered when the sub-realization does not meet the conditions of any of the cases. Case 4 is a slight modification of the Erdos Gallai condition, as stated below, that checks if a sub realization is graphical or not. Specifically, a sequence d is graphical if and only if

1. $\sum_{i=1}^{k} d_i$ is even
2. $\sum_{i=1}^{r} d_i \leq r(r-1) + \sum_{k=r+1}^{n} \min\,(r, d_k)$ where $k \in rightSubset$.

This case is entered if the nodes in the graph are pairwise adjacent.

Consider the example of generating a graph on 6 nodes with the degree sequence [3,2,2,1,1,1].

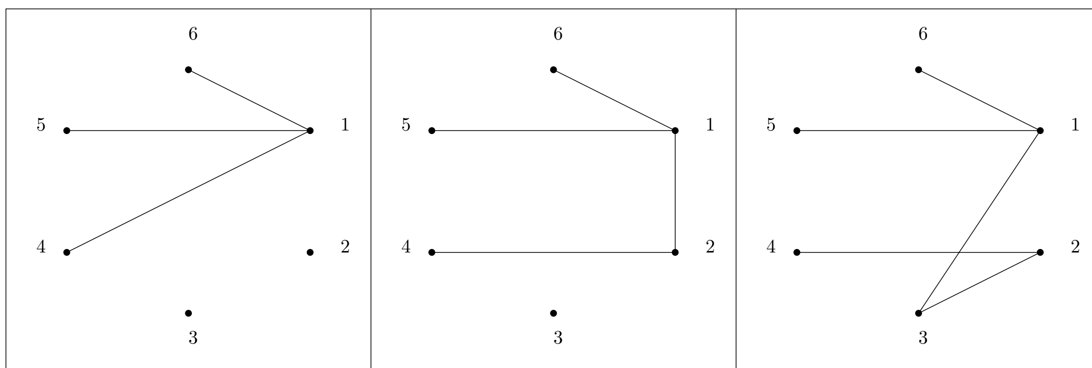


Figure 2.7: Steps in generating a graph of degree sequence [3,2,2,1,1,1] using Tripathi's method; (i) r=1; (ii) r=2; (iii) r=3

At $r = 1$, i.e., node 1 selected as the hub node, nodes are available from the right subset, thereby satisfying the conditions for case 0. This case enables r to be connected to nodes 4, 5, and 6 randomly picked from the list of available nodes, i.e., 2, 3, 4, 5, 6. This saturates node 1 thereby resulting in a residual degree sequence [0,2,2,0,0,0], and r is incremented to 2.

When r is 2, node 3 is seen to be the only available node. Hence 2 gets connected to 3. Node 2 is still deficient by 1 and hasn't been connected to any of the nodes in the left subset. This situation qualifies for case 1, particularly the second sub-case since the deficiency is 1. The edges (1,4) and (2,3) are replaced with (1,2) and (2,4). This then saturates node 2, resulting in a residual sequence [0,0,1,0,0,0], and therefore we increment r to 3.

When r is 3, we find that there are no more nodes left to join to r, and node 3 hasn't been connected to any other node. This scenario fits case 1, specifically the first sub-case since the deficiency is 2. The edge (1,2) is replaced by (1,3) and (2,3). The graph generated as a result has all the nodes fully saturated with the nodes realizing inputted degree sequence. This concludes the generation process. The algorithms for the different cases are as follows.

Case 0:

Algorithm 2: CASE0(parameters: r,Seq,$rightSubset$)

Data: r: Hub node, Seq: Degree Sequence, $rightSubset$: $[r+1:n]$

Result: $G(V,E)$: Updated graph, Seq: Updated degree sequence

```
1 for i ∈ rightSubset do
2     if i and r are not saturated AND (r,i) ∉ E then
3         Add (r,i) to E;
4         update Seq
5     end
6 end
7 return G,Seq
```

Case 2:

Algorithm 3: CASE2(parameters: r,Seq,$leftSubset$)

Data: r: Index of the node being saturated, Seq: Degree Sequence, $leftSubset$: $[0:r]$

Result: $G:(V,E)$: Updated graph, Seq: Updated degree sequence

```
1 for i ∈ leftSubset AND k ∈ rightSubset AND u from 0,...,n do
2     if u! = r AND AND (i,u),(r,k) ∈ E AND (u,r),(i,k) ∉ E then
3         Remove (i,u) from E;
4         Add (u,r) and (i,k) to E;
5         update Seq;
6         return G,Seq
7     end
8 end
9 return G,Seq
```

Case 1:

Algorithm 4: CASE1(parameters: r,Seq,$leftSubset$)

Data: r: Hub node, Seq: Degree Sequence, $leftSubset$: $[0 : r]$

Result: $G(V, E)$: Updated graph, Seq: Updated degree sequence

```
1  i ← i : i ∈ leftSubset and (i, r) ∈ E;
2  if Seq[r] ≥ 2 then
3  |   for u from 0, . . . , n do
4  |   |   if u! = r AND (i, u) ∈ E AND (r, u) ∉ E then
5  |   |   |   Remove (i, u) from E;
6  |   |   |   Add (r, i) and (r, u) to E;
7  |   |   |   update Seq;
8  |   |   |   return G,Seq
9  |   |   end
10 |   end
11 else
12 |   if Seq[r] == 1 then
13 |   |   for u from 0, . . . , n AND k ∈ rightSubset do
14 |   |   |   if u! = r AND (r, k), (i, u) ∈ E AND (r, u) ∉ E then
15 |   |   |   |   Remove (i, u) from E;
16 |   |   |   |   Add (r, u) and (i, k) to E;
17 |   |   |   |   Update Seq;
18 |   |   |   |   return G,Seq
19 |   |   |   end
20 |   |   end
21 |   end
22 end
23 return G,Seq
```

Case 3:

Algorithm 5: CASE3(parameters: r,Seq,$leftSubset$)

Data: r: Index of the node being saturated, Seq: Degree Sequence, $leftSubset$: $[0:r]$

Result: $G:(V,E)$: Updated graph , Seq: Updated degree sequence

```
1  for i, j ∈ leftSubset AND u, w ∈ rightSubset do
2      if i! = j AND (j, w), (i, u) ∈ E AND (i, j), (u, r) ∉ E then
3          Remove (u, i), (j, w) from E;
4          Add (u, r), (i, j) to E and update Seq;
5          if r is saturated then
6              return G,Seq
7          end
8      end
9  end
10 return G,Seq
```

Case 4:

Algorithm 6: CASE4(parameters: r,DSeq,Seq

Data: r:Hub node, Seq: Degree sequence, $leftSubset$: [0:r-1], $rightSubset$: $[r+1{:}n]$

Result: Result of graphicality test for sub-realization

```
1  k ←size(Seq);
2  if sum(Seq)%2!=0 then
3      return False
4  end
5  RHS ← r(r − 1);
6  for j ∈ rightSubset do
7      RHS+ =min(r,Seq[j])
8  end
9  LHS ←sum(leftSubset);
10 if LHS ≤ RHS then
11     return True
12 else
13     return False
14 end
```

Complexity

Tripathi's method for generating graphs given a degree sequence is to generate graphs as long as the inputted sequence is graphical. The different cases and results in a generated graph cover all the scenarios and obstacles encountered during the process.

Tripathi's algorithm constructs a graph by building sub-realization. Even though the sub-realizations get easier to compute as the algorithm builds on, the maximum number of steps is $\sum_{i=1}^{n} d_i$. For the worst case, $\sum_{i=1}^{n} d_i$ is $O(n^2)$. The total complexity is hence computed as $O(n \sum d_i)$, which is $O(n^3)$.

Chapter 3

Special cases

The following sections cover generation methods for a few special cases devised, with the principles acquired from the ideas of graph generation. There are two special cases, both of which are subtypes of graphs.

3.1 Special case: Forests

A tree can be defined as a graph without cycles. A forest can be seen as a collection of trees. Given a set of non increasing positive integers say $d_1, d_2, \ldots, d_n$, a forest with n vertices realizing these degrees exist if and only if the following conditions are satisfied:
1. $\sum_{i=1}^{n} d_i$, is even
2. $\sum_{i=1}^{n} d_i \leq 2(n\text{-}1)$

These conditions ensure that the inputted sequence of integers is realizable as a forest.

The proof for these conditions is constructive. We propose an incremental algorithm that constructs a forest for a given degree sequence if one exists.

Once the inputted sequence is confirmed to be realizable, the generation ensues. Along with the data structure to maintain the degree sequence, there is also an array to maintain the visit status of each node. The visit status is 1 if the node has been visited and 0 if the node has never been visited. This makes sure that no cycles are created during the process.

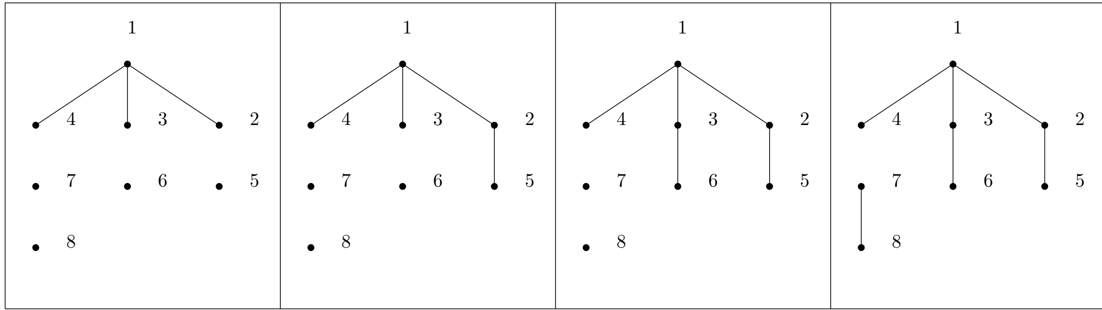


FIGURE 3.1: Steps in generating a forest realizing [3,2,2,1,1,1,1,1]

To understand the process better, the following is an example of how the generation would go to generate a forest on 8 nodes having the degree sequence [3,2,2,1,1,1,1,1]. Figure 3.1 shows the different stages in the generation process. Initially, all the nodes are marked as unvisited.

We first pick the first unsaturated node, i.e., node 1 in this case. We then start joining this node to the available unvisited nodes in their decreasing order of degrees 2, 3, and 4 in this case. We mark the nodes as visited and then introduce edges (1,2), (1,3), and (1,4). 1 and 4 are fully saturated at this stage.

We then saturate the nearest unsaturated node in the list, i.e., node 2. On looking for an available unvisited node, we arrive upon node 5 and hence introduce edge 2-5 and mark 5 as visited. This fully saturates nodes 2 and 5.

The next nearest unsaturated node in the list is node 3. We look for an available unvisited node and therefore connect 3 to 6 and mark 6 as visited. Nodes 3 and 6 become fully saturated.

Node 7 is found to be the nearest unsaturated node. We look for an available unvisited node and therefore connect 7 to 8 and mark 8 as visited, making both 7 and 8 fully saturated. This leaves us with the desired forest representing the inputted degree sequence, therefore, terminating the process.

Complexity

To check if the given degree sequence is realizable as a forest, the time taken is $O(n)$. In the worst case, a node takes $O(n)$ to get saturated. All the n nodes take $O(n^2)$ to get saturated. The total complexity can be calculated as $O(n+n^2)$, which is $O(n^2)$.

The algorithm can be stated as follows.

Algorithm 7: ForestGen(parameters: n,Seq, F)

```
   Data: n: Number of nodes, Seq: Degree Sequence, F: Forest
   Result: Forest F : (V, E) realizing Seq
1  vis ← n∗[0];
2  i ←0;
3  while Seq contains unsaturated nodes do
4      for j from i + 1, . . . , n do
5          if  i and j are unsaturated AND vis[i] ==0 AND vis[j] ==0 then
6              Add (i, j) to E and update Seq;
7              vis[j]←1;
8          end
9      end
       // Finding the nearest unsaturated node to assign as hub node
10     for j from i + 1, . . . , n do
11         if j is unsaturated then
12             break
13         end
14     end
15     i ← j;
16 end
17 return F(V, E)
```

3.2 Special case 2: Split Graphs

An undirected graph $G = (V, E)$ can be considered to be a split graph if and only if [10] the vertex set V can be divided into a clique C and an independent set I i.e. $V = C \cup I$.

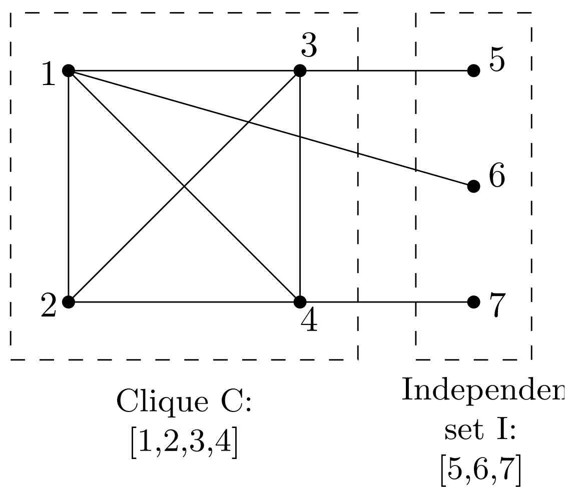


Figure 3.2: Split graph example

In Figure 3.2,

- Clique $C = \{1, 2, 3, 4\}$
- Independent set $I = \{5, 6, 7\}$

3.2.1 Applications of split graph

Split graphs have applications in Social Network Analysis. Figure below demonstrates the specific use case of the privacy policy on Facebook.

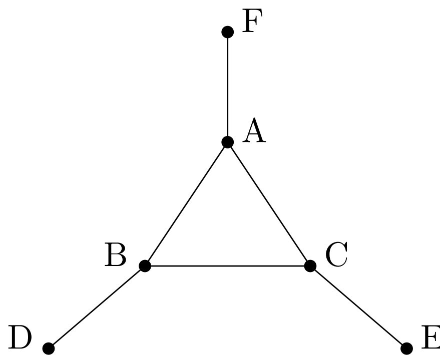


Figure 3.3: Split graph social network detection

Figure 3.3 denotes the following relationship. Assume that F allows for their friends and friends of friends to view their content on Facebook. Let A, B, and C be friends. F is a friend of A, D is a friend of B, and E is a friend of C. If F shares something, then A, B, and C can view it but not D and E.

Relationships of such nature can be visualized and represented using split graphs. These visualizations can aid in quantifying and executing the required functionalities.

The following are some characteristics of a valid split graph G:

1. G is a split graph if only if $\overline{G}$ is also a split graph

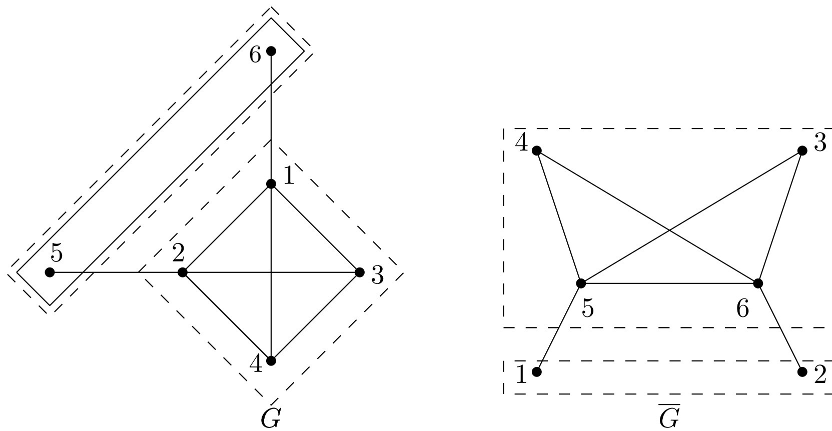


FIGURE 3.4: Split graph characteristics: Split graph G and Split graph $\overline{G}$

2. All split graphs are chordal since split graphs $\subset$ chordal graphs [11].

3. If G is a split graph, then every graph with the same degree sequence is also a split graph. Figure 3.5 shows the two different split graphs realizing the same degree sequence [4,4,3,3,1,1].

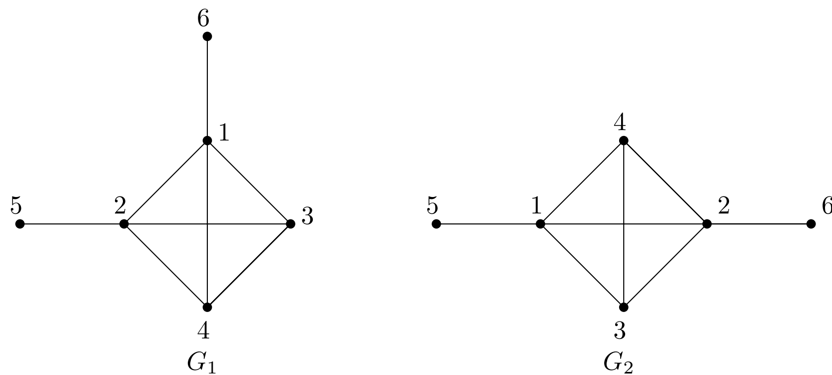


FIGURE 3.5: Split graph characteristics: Multiple split graphs with degree sequence [4,4,3,3,1,1]

3.2.2 The Recognition algorithm

Determining the split point

The split point m plays an important part in the algorithm since it's crucial to verify if a degree sequence is realizable as a split graph. For a graph G on n nodes, with the indices of the nodes i ranging from 1 to n and degree sequence $d_1 \geq d_2 \geq \ldots \geq d_n$, the split point m can be defined as the highest node index that meets the condition $d_m \geq m - 1$ where $1 \leq m \leq n$. This split point m can be determined by iterating from 1 to n and stopping when the condition is violated.

Verifying the condition

Once the split point m is determined, G can be considered to be a split graph if and only if $\sum_{i=1}^{m} d_i = m(m-1) + \sum_{i=m+1}^{n} d_i$

This split graph condition can be used to determine if the split graph can be generated.

Generating the graph

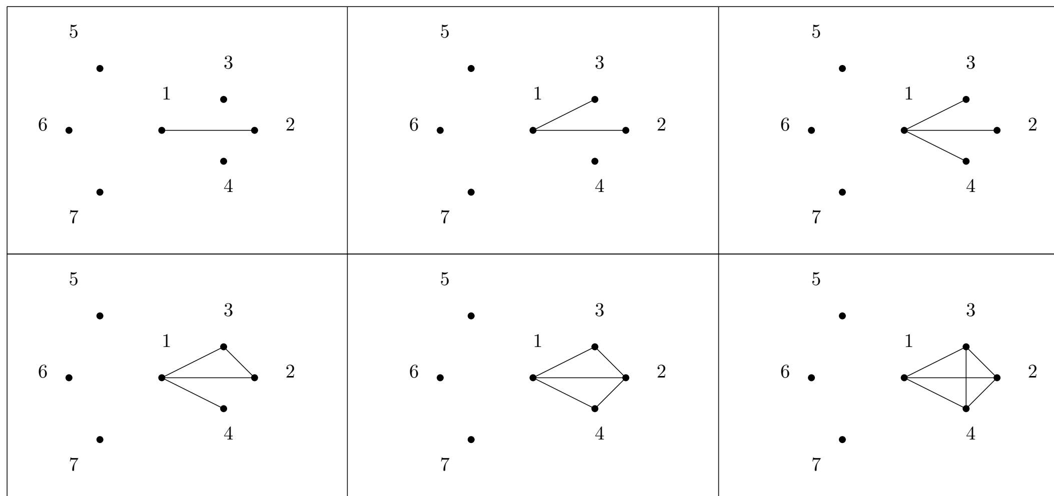


FIGURE 3.6: Generation of clique

If the condition is verified, the generation starts. The generation takes place in two stages. The first stage involves the generation of the connected component, i.e., the clique. This is then followed the connecting the remaining nodes to an available node from the independent set.

This can be understood using an example to generate a split graph on 7 nodes, with degree sequence [6,3,3,3,1,1,1]. If the node labels are 1,...,n, the split point m is calculated as 4. This determines the clique set [1,2,3,4] and the independent set to be [5,6,7].

The connected set is built using a basic generation process [12] using two loops to iterate through the nodes in the clique set. This leads to the edges (1,2), (1,3), (1,4), (2,3), (2,4), and (3,4) which ultimately builds the graph in Figure 3.6.

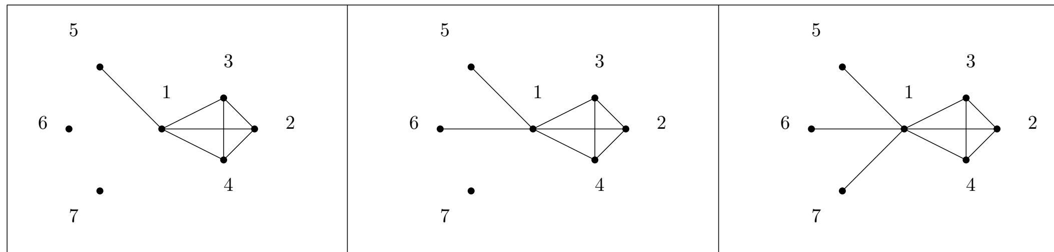


FIGURE 3.7: Connecting nodes of the independent set to the clique

The randomization happens in the steps following the generation of the connected component. In this case, node 1 happens to be the only available node, which then leads to edges (1,5), (1,6), and (1,7), thereby generating the entire split graph as in Figure 3.7.

The problem of counting the number of labeled split graphs is equivalent to counting (0,1) matrices given marginals (row sum and column sum)[13]. The generation process of a split graph $G : (V, E)$ finds the split point m, and first builds the clique C. As mentioned before, the connection of the nodes from the independent set I to the clique C, is where the randomness comes from. The number of ways of connecting nodes from I to C can be thought of as the number of (0,1) matrices. The marginals, in this case, would be the degree of the nodes in set I.

The algorithm for generating split graphs is formally stated as below:

```
Algorithm 8: SplitGraphGen(parameters: n,Seq, SG)
   Data: Seq: Degree Sequence, n: number of nodes
   Result: SG(V,E): Split graph for the sequence
 1 LHS, RHS, m ←0;
 2 for i from 1, ..., n do
 3 |   if Seq[i] ≥ i-1 then
 4 |   |   m ← i;
 5 |   end
 6 |   else
 7 |   |   break;
 8 end
 9 LHS ← sum(Seq[1:m]);
10 RHS(m-1)+ sum(Seq[m + 1:n]);
11 if LHS == RHS AND m ≤ n then
12 |   for i from 1,...,m do
13 |   |   for j from i + 1, ..., m do
14 |   |   |   Add (i,j) to E and update Seq;
15 |   |   end
16 |   end
17 |   for i from m + 1, ..., n do
18 |   |   availableCliqueNodes ←[];
19 |   |   for j from 1, ..., m do
20 |   |   |   if j is not saturated and i! = j and (i, j) ∉ E then
21 |   |   |   |   Append j to availableCliqueNodes
22 |   |   |   end
23 |   |   end
24 |   |   while i is not saturated do
25 |   |   |   j ←random node picked from availableCliqueNodes;
26 |   |   |   Add (j, i) to E and update Seq;
27 |   |   |   Remove j from availableCliqueNodes;
28 |   |   end
29 |   end
30 |   return SG(V,E)
31 end
```

Chapter 4

Exhaustive methods

In this chapter, we explore some exhaustive methods that are aimed at generating labeled graphs exhaustively. A degree sequence can have multiple graphs that represent it. The problem of counting these graphs is an open problem that is still being studied. We will be discussing two exhaustive algorithms in this chapter. The first one being a method proposed by James Riha [4], and the second one is a method that was proposed by Kim[5]. The degree sequences in the figures to come are represented using 2D arrays wherein the first row D stands for the degree at each node, and the second row N stands for node labels.

4.1 James Riha exhaustive algorithms

The James Riha algorithm [4] uses equivalence classes to guide the generation process. An equivalence class in this context is a subset of nodes that have the same degrees. The nodes are divided into different equivalence classes based on their degree distribution.

James and Riha prescribe a method that generates different ways to pick nodes from the different equivalence classes. The ways in which the nodes are picked are referred to as solutions. The nodes are chosen for connections according to the different solutions generated. For example, if there are two equivalence classes $Eq0$ and $Eq1$ both containing two nodes, and the degree of the hub node is 2, then the different solutions are [2,0], [0,2], and [1,1]. The solution [2,0] means that two nodes are picked from $Eq0$ and none from $Eq1$.

The algorithm has been executed recursively. Each solution, when explored separately, can either yield a successful graph or a failure. The failure is detected when the number of nodes other than the hub node is lesser than the degree of the hub node (indication of a non-realizable degree sequence). If the path is successful, then the graph is displayed, and if a failure is encountered, we recurse back to the point of decision making, i.e., recent solution choice, to explore the other solutions in line. This forms the base case for the recursive algorithm. The repeated problem of the recursive algorithm can be described as follows.

1. The node with the highest degree is chosen as the hub node
2. The rest of the nodes are divided into different equivalence classes based on their degrees, i.e., nodes of the same degree are placed in one equivalence class
3. Nodes can be picked in different ways from the equivalence classes, which can be inferred from the different solutions generated.
4. The different solutions are iterated one by one, and the hub node is joined to nodes according to the solution.

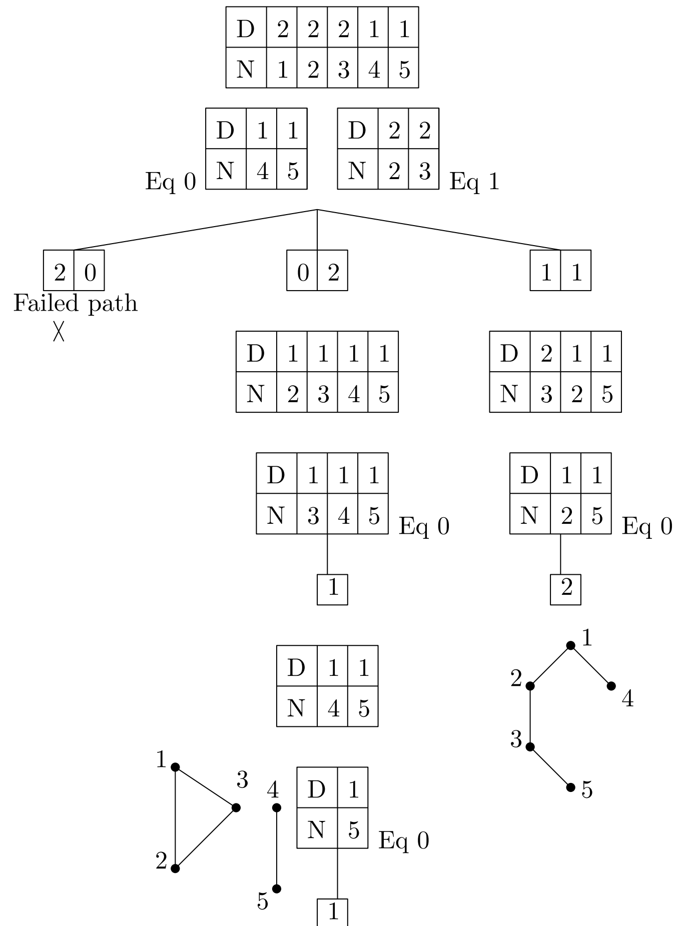


FIGURE 4.1: Steps in generating graphs using James Riha algorithm;

To regulate the generation process, a standardization is introduced, which requires us to pick the nodes from the equivalence classes in the order of their labels.

Consider the example of generating a graph on 5 nodes with the degree sequence [2,2,2,1,1]. Figure 4.1 demonstrates this process. Node 1 is selected as the hub node. The rest of the nodes with non-zero degrees are taken and placed in different equivalence classes depending on their residual degrees. Nodes 2, 3, 4, and 5 get divided into two equivalence classes $Eq0$ (with nodes 4 and 5 of degree 1) and $Eq1$ (with nodes 2 and 3 of degree 2). Since the degree of the hub node is 2, two nodes need to be selected from the two equivalence classes to saturate it. There are three solutions as a result, namely [2,0], [0,2], and [1,1] (each element denoting the number of elements to be picked from the equivalence class). We explore the solutions one by one.

Exploring Solution [2,0] would result in edges (1,4) and (1,5), thereby leading to a non-graphical residual degree sequence. The path is indicated as a failed path, and we return to pick the next solution in line, i.e., solution [0,2], which leads to edges (1,2) and (1,3) that yield a realizable degree sequence, which allows us to continue further down that path.

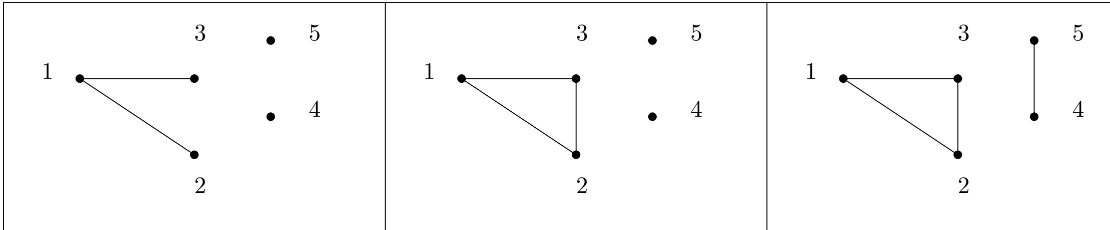


FIGURE 4.2: Graph 1 generated by exhaustive James Riha algorithm

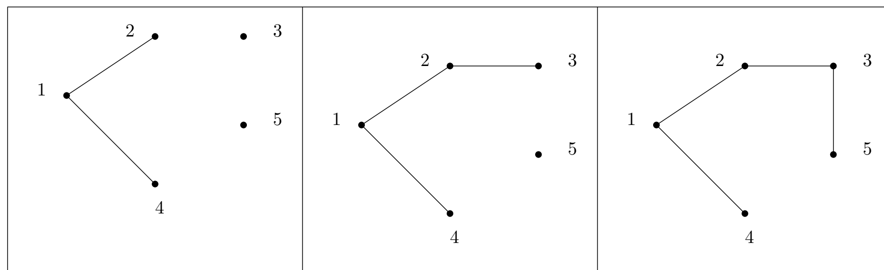


FIGURE 4.3: Graph 2 generated by exhaustive James Riha algorithm

On continuing further, node 2 gets picked as the next hub node. The rest of the nodes all have the same degrees, which implies that all the nodes get placed into a single equivalence class. This has one solution [1], which results in edge (2,3) and a realizable sequence that allows us to recurse further down.

Node 4 gets picked next as the hub node, leading to another singular solution, followed by an edge (4,5), thereby arriving at an all-zero degree sequence (realizable) and a final graph for that path as seen in Figure 4.2. Similarly, the final solution [1,1] is explored, leading to a successful path(the graph is as in Figure 4.3) and a failed path as shown in Figure 4.1. The algorithms are described below:

The algorithm is guaranteed to generate at least one correct graph amidst the multiple failed paths. This is because Hakimi's way of generation is implicitly included in the multiple solutions that are generated.

Algorithm 9: ExhJRgeneration(parameters: *Seq*)

```
   Data: Seq: 2D array [[Node label, Degree],...]
   Result: Result of whether path results in failure (False) or successful (True)
1  Global variables: G(V,E)- Graph being generated;
2  Remove zero degree nodes Seq and sort Seq according to the degrees;
   // Hub will be Seq[1] i.e. first element
3  if All nodes are saturated then
4  |  Display G ;return True
5  eqc ← list of nodes in Seq[2:] grouped by their degrees;
6  if len(eligibleNodes) < Hub degree then
7  |  return False
8  else
9  |  eqcLen ← list containing lengths of each eq class;
10 |  sol ← Call jrsplit(Seq[0][1], len(eqc), eqcLen);
11 |  for i from 1,...,len(sol) do
12 |  |  for j from 1,...,len(eqc) do
13 |  |  |  for k from 1,...,sol[i][j] do
14 |  |  |  |  Add (Hub node, eqc[j][k]) to E and update Seq;
15 |  |  |  end
16 |  |  end
17 |  |  Call ExhJrgeneration(Seq);
18 |  |  Undo edges to move on to next solution;
19 |  end
```

Algorithm 10: jrsplit(parameters: *HubDegree*,*nEqc*,*eqcLen*)

Data: *HubDegree*, *nEqc*:Number of equivalence classes, *eqcLen*:Size of the equivalence classes

Result: *sol*: Different ways of picking nodes

```
1  x ←=[0]* nEqc; sol ←[];
2  if HubDegree >sum(eqcLen[1:nEqc]) then
3  |   return
4  k ← eqcLen[1:nEqc]; flag ←1;
5  while flag do
6  |   if k==0: then
7  |   |   x[k]←0;
8  |   |   while x[k]==0 AND k < nEqc do
9  |   |   |   k++
10 |   |   end
11 |   |   if x[k]==0 then
12 |   |   |   break;
13 |   |   end
14 |   end
15 |   while HubDegree ≥sum(eqcLen[1:k]) do
16 |   |   x[k]←0;
17 |   |   while x[k]==0 and k < nEqc do
18 |   |   |   k++
19 |   |   end
20 |   |   if x[k]==0 then
21 |   |   |   flag ←0; break;
22 |   |   end
23 |   end
24 |   HubDegree++; x[k]–; k–;
25 |   while nEqc > eqcLen[k] do
26 |   |   x[k]← eqcLen[k]; nEqc -= eqcLen[k]; k–;
27 |   end
28 |   x[k]← eqcLen; nEqc-=x[k]; Append x to sols;
29 end
30 return sol
```

4.2 Kim's exhaustive algorithm

The second exhaustive algorithm is a method by Kim [5]. In Kim's first algorithm, there are no failures, unlike the previously discussed James Riha method. Other differences will be discussed in the sections to follow.

The centrality of the algorithm lies in building the rightmost adjacency set for every hub node chosen. In a set of nodes arranged in non-increasing order of their degrees, the leftmost adjacency set is the set of the highest degree nodes that can saturate a given hub node.

The leftmost adjacency set always preserves graphicality, and picking nodes beyond that is when failure starts happening. However, the rightmost adjacency set refers to the lowest degree nodes that can be joined to the hub node to saturate it without breaking graphicality.

The rightmost adjacency set is built by iterating nodes from right to left and checking if temporary connections between the iterated node and the hub node break graphicality. The graphicality tests done to check if a node is eligible for the rightmost adjacency set are referred to as CG tests (constrained graphicality tests).

According to Kim[5], the smaller sets i.e. the set of nodes to the left of the rightmost adjacency set will not break graphicality either, and can formally be stated as: For two adjacency sets $A(i) = \{\ldots, a_k, \ldots\}$ and $B(i) = \{\ldots, b_k, \ldots\}$, $B(i) \leq A(i)$ if and only if $b_k \leq a_k$ for all $1 \leq k \leq d_i$.

After building the rightmost adjacency set, other adjacency sets that are smaller (i.e., smaller sets) are built, each of which is individually picked to generate a successful graph. Kim's algorithm has been executed through a recursive algorithm where the common operation is:

1. Sort the degree sequence in the decreasing order of degrees
2. Assign the first node in the sorted list as the hub node (the rest of the nodes are considered to be the available nodes)
3. Calculate the rightmost adjacency set in the following manner. Begin traversing the available nodes one by one from right to left, and perform a CG test. The traversed node, if eligible, is appended to the rightmost adjacency set (which is initially empty for each calculation). The size of the rightmost adjacency set is

the degree of the hub node

4. Build smaller sets with respect to the rightmost adjacency set.
5. Connect hub node to nodes prescribed by the smaller set

Consider an example of generating a graph on 5 nodes with the degree sequence [2,2,2,1,1]. The different steps in the generation can be seen in Figure 4.4. We start by choosing 1 as the hub node. We build the rightmost adjacency set by iterating from the lowest degree node 5, performing CG tests on the nodes, and adding them until node 1 is saturated.

The rightmost adjacency set is calculated as [3,5]. We now construct sets that are to the left of the rightmost adjacency i.e. [[2,3], [2,4], [2,5], [3,4], [3,5]]. Since this is an exhaustive algorithm, all the smaller sets are explored separately, and graphs are displayed accordingly. On picking [3,5], edges (1,2) and (1,3) are introduced, thereby fully saturating 1 and resulting in a residual degree sequence of [1,1,1,1].

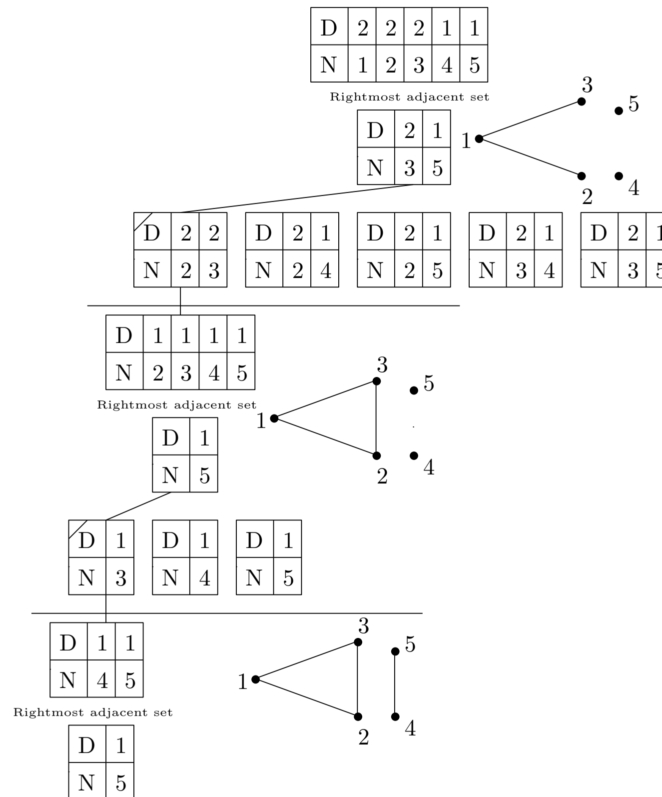


FIGURE 4.4: Example of one of the graphs generated by the Kim exhaustive algorithm

We process the node list by discarding the zero elements at every step and sorting them in the decreasing order of their degrees. The process recursively takes place for the next hub node, i.e., node 2, resulting in [5] to be calculated as the rightmost adjacency set and [[3],[4],[5]] as the smaller sets. Continuing down the smaller set [3] leads to an

edge (2,3) and residual degree sequence [1,1]. Node 4 is the next hub node picked, for which [5] gets calculated as the rightmost adjacency set for which [[5]] is the only smaller set. Therefore we add an edge (4,5) and arrive at the end of that path since the residual degree sequence is empty.

We then backtrack to the latest fork and choose a different smaller set if it exists. If there are no smaller sets at the fork, we backtrack again. The entire process is terminated once every smaller set generated has been chosen. The complete set of graphs generated in this run can be seen in Figure 4.5.

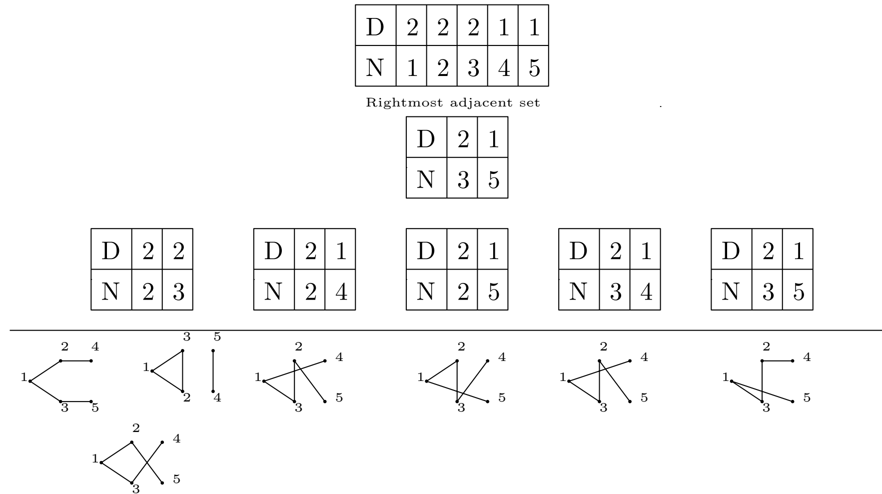


FIGURE 4.5: The collection of graphs generated by exhaustive Kim algorithm

---

Algorithm 11: BuildSmallerSets(parameters: $NodeList$,$Rightmost$)

Data: $NodeList$: List of eligible nodes, $Rightmost$: rightmost adjacency set

Result: $SmallerSets$: Collection of adjacency sets that are $\leq Rightmost$

1 $RangeSmallerSets \leftarrow$ range(min($NodeList$), $\ldots$, $Rightmost[-1]$);

2 $SmallerSets \leftarrow$ List(combinations($RangeSmallerSets$,len($Rightmost$)));

3 **for** i *from* $0, \ldots, len(SmallerSets)$ **do**

4   **for** j *from* $0, \ldots, len(Rightmost)$ **do**

5   **if** $SmallerSets[i][j] > Rightmost[j]$ **then**

6   remove $SmallerSets[i]$ from $SmallerSets$; break

7   **end**

8   **end**

9 **end**

10 **return** $SmallerSets$

Algorithm 12: exhKimRecGen(parameters: *Seq*)

```
   Data: Seq: 2D array [[Node label, Degree],...]
 1 Global variables: G(V,E)- Graph;
 2 Remove zero degree nodes from Seq and sort list in decreasing order of the
     degrees;
   // Hub will be Seq[1] i.e. first element
 3 NodeList ← nodes from Seq; n ← Number of nodes;
 4 if All nodes are saturated OR Seq is empty then
 5 |   Display G; return;
 6 else
 7 |   Rightmost ←[];
 8 |   for i from n,...,1 do
 9 |   |   if len(Rightmost) < Hub degree then
10 |   |   |   Temporarily add (Hub node,Seq[i][0]) to E and update Seq;
11 |   |   |   if C.G Test(Seq)== True then
12 |   |   |   |   Append Seq[i][0] to Rightmost
13 |   |   |   Remove (Hub node,Seq[i][0]) from E and update Seq;
14 |   |   else
15 |   |   |   break;
16 |   end
17 |   Sort Rightmost;
18 |   SmallerSetIndices ←Call BuildSmallerSets(NodeList, Rightmost);
19 |   SmallerSets ← Translate the indices in the SmallerSetIndices into node
   |     labels;
20 |   for i in SmallerSets do
21 |   |   for j in i do
22 |   |   |   Add (Hub node,j) to E and update Seq;
23 |   |   end
24 |   |   Call exhKimRecGen(Seq);
25 |   |   for j in i do
26 |   |   |   Remove (Hub node,j) from E and update Seq;
   |   |   |   // This reset is to be able to process the different solutions
27 |   |   end
28 |   end
```

4.3 Comparing the exhaustive algorithms: James Riha exhaustive algorithm VS Kim-1 exhaustive algorithm

This section compares both the exhaustive algorithms based on the number of graphs generated, isomorphic repetitions, rejections, and running time.

4.3.1 Number of graphs generated

In the James-Riha exhaustive algorithm, we regulate how nodes are picked from the equivalence classes (in the order of node labels) since the algorithm aims to reduce isomorphic repetitions. This leads to the generation of a subset of labeled graphs. Consider the example of generating a graph on 6 nodes with degree sequence [3,3,2,2,2,2].

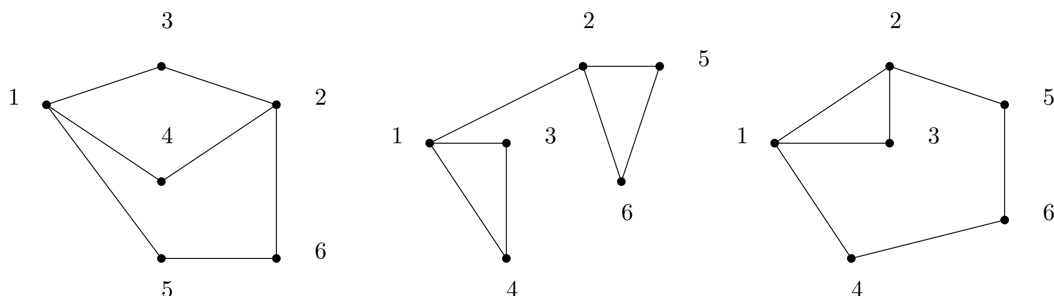


FIGURE 4.6: Graph generated by James Riha exhaustive algorithm for degree sequence (3,3,2,2,2,2)

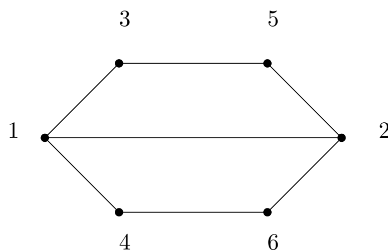


FIGURE 4.7: Graph missed out by James Riha algorithm

The graphs in Figure 4.6 are the only graphs that are generated by the James Riha method. The graph in Figure,4.7 however, gets missed out by the algorithm. This shows that the James-Riha algorithm does not generate graphs from all isomorphic classes.

Kim's exhaustive algorithm, on the other hand, generates a larger subset of labeled graphs (if not the entire set of labeled graphs), thereby generating graphs from more isomorphic classes compared to the James-Riha exhaustive method. For instance, the

same degree sequence [3,3,2,2,2,2], when inputted to Kim's exhaustive method, generates 54 graphs as opposed to the James-Riha method that generates only 3 graphs.

4.3.2 Isomorphic repetitions

In graph theory, an isomorphism of graphs G and H is a bijection between the vertex sets of G and H such that any two vertices u, $v \in G$ are adjacent in G, if and only f(u), f(v) are adjacent in H. Exhaustive methods can tend to produce graphs with a lot of isomorphic repetitions. The James-Riha algorithm aims to reduce isomorphic repetitions, as a result of which there are minimal isomorphic repetitions.

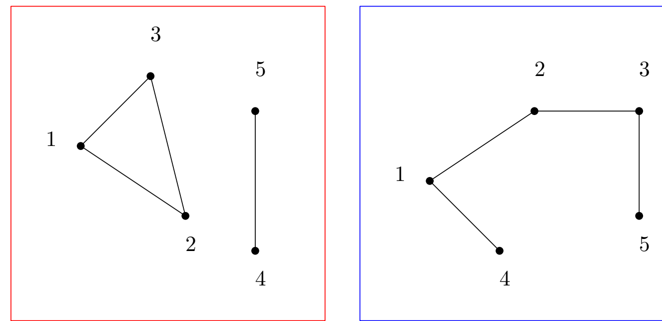


FIGURE 4.8: Graph generated by James-Riha exhaustive algorithm for degree sequence (2,2,2,1,1)

Figure 4.8 shows the graphs generated by the James-Riha exhaustive method for the degree sequence (2,2,2,1,1). The algorithm produces only two graphs, and both belong to different isomorphic classes, denoted by blue and red circles.

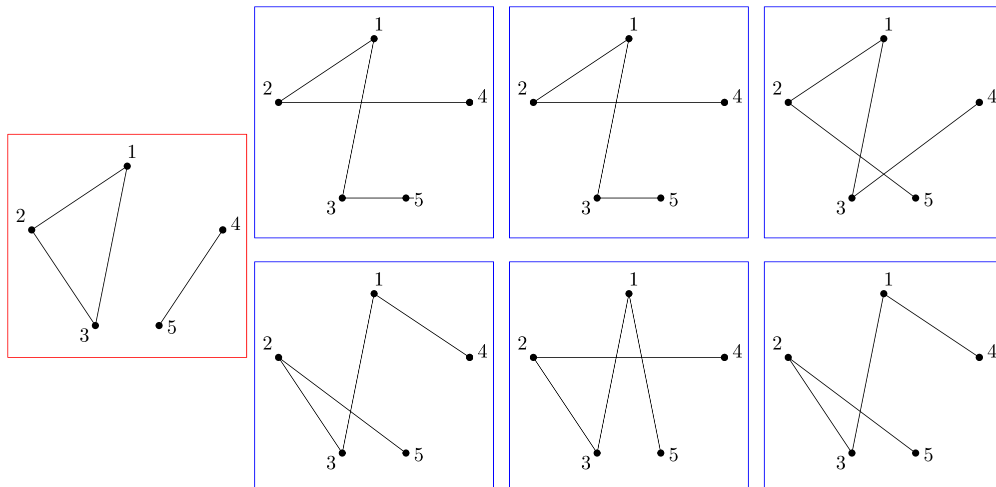


FIGURE 4.9: Graph generated by Kim's exhaustive algorithm

On the other hand, when run for the same input degree sequence, Kim's exhaustive algorithm produces graphs with a lot of isomorphic repetition. Figure 4.9 for example,

shows 6 graphs from the blue isomorphic class and 1 graph from the red isomorphic class.

4.3.3 Rejections

In terms of rejection, the James Riha algorithm faces many rejections since not all solutions lead to realizable degree sequences.

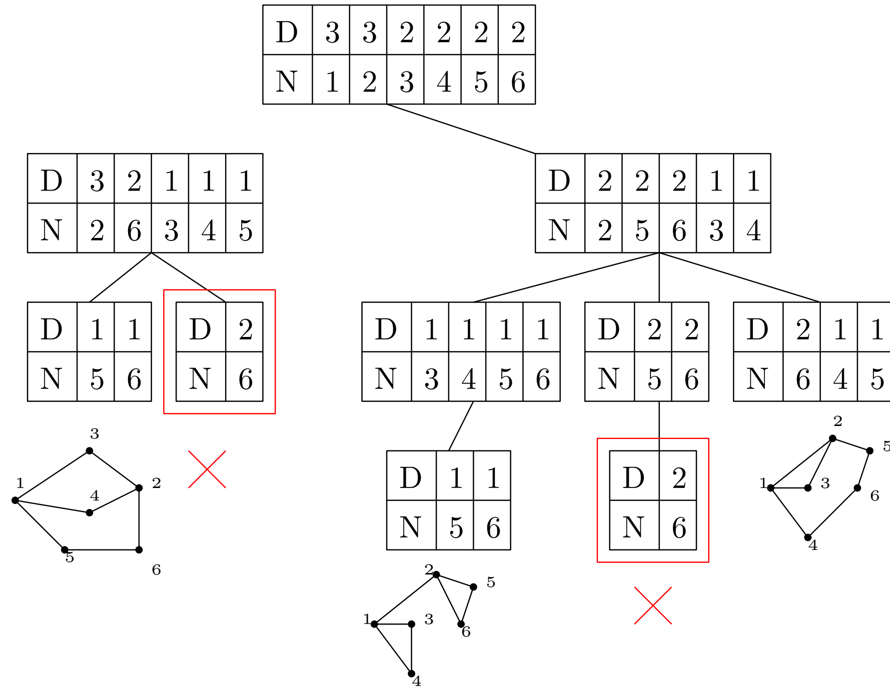


FIGURE 4.10: Rejections faced by the James-Riha exhaustive method

As seen in Figure 4.10, the James-Riha algorithm deals with a couple of paths that lead to failures. This requires the algorithm to stop pursuing that path altogether, followed by a backtrack to the split point. This also leads to a lot of overhead due to the lack of predictability of these failures in earlier stages.

Figure 4.11 on the other hand, shows Kim's exhaustive algorithm generating multiple graphs without facing any failures or rejections along the way. There are zero rejections because the method operates by selecting a fail-proof rightmost adjacent set to generate smaller sets.

Both the exhaustive algorithms described in this chapter have their pros and cons. The James Riha exhaustive algorithm can be utilized if the purpose is to produce graphs with the least possible isomorphic repetition. Kim's exhaustive algorithm can be utilized if the aim is to generate as many graphs as possible.

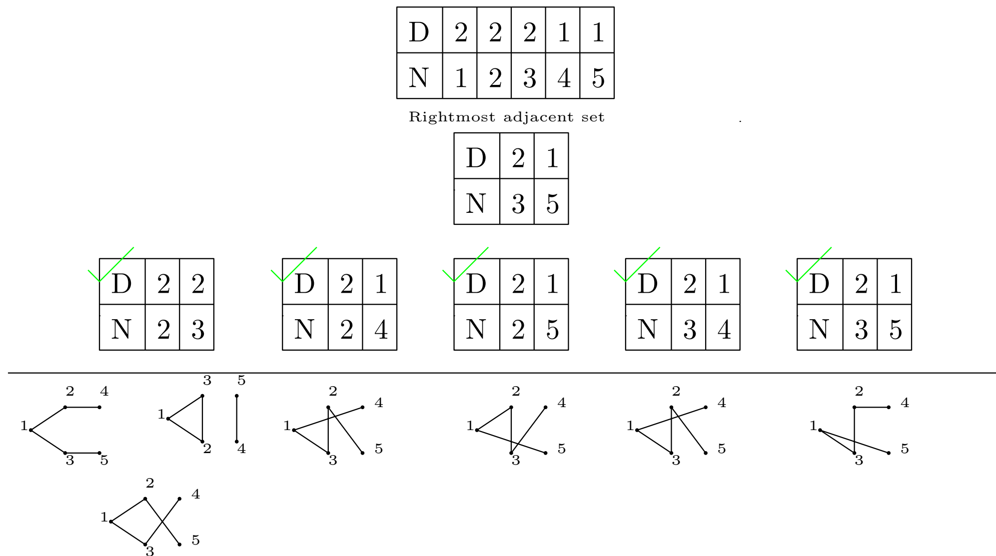


FIGURE 4.11: Rejection free Kim's exhaustive method

In terms of efficiency, the James Riha exhaustive algorithm does not require any graphicality tests to determine if a path succeeds. The success of a path is determined simply by checking if the number of nodes in the sequence (excluding the hub node) is enough to saturate the hub node. Kim's exhaustive algorithm, however, has multiple constrained graphicality tests for each hub node that is selected. This overhead ultimately adds up for the numerous smaller sets that are pursued. The implications are discussed in the next chapter.

Chapter 5

Sampling methods and comparative studies

5.1 Modification of the exhaustive methods into sampling methods

The above methods that have been proposed can generate exhaustively from a limited space of labeled graphs. Moreover, the previous sections demonstrate how innumerable the number of labeled graphs can be, even for simple cases with fewer nodes and edges. The exhaustive listing becomes intensive when the size of the graphs increases with the increasing degree sequence.

Sampling algorithms generate a single labeled graph randomly picked from the space of multiple labeled graphs, satisfying a given degree sequence. The problem of generating graphs labeled or unlabeled of arbitrary sizes uniformly at random is one that's been studied for years. The problem also leads into the counting problem, i.e., the number of graphs for a given degree sequence.

Kim explains that the number of graphs $G(D)$ that realize a given degree sequence D increases quickly with the number of nodes N. An upper bound $|G(D)| \leq \prod_{i=0}^{N-1} d_i!$ on the number of graphs that can be generated for such degree sequences is given[14]. But this upper bound holds only for short degree sequences, making sampling necessary for modeling real-world complex networks.

Sampling a single graph at random from a space of all graphs that satisfy a specific

degree sequence has numerous applications in many fields. The next section reveals details about why sampling is effective and a few practical applications.

5.2 Application of sampling algorithms

Real-life applications that require graphs to be generated need the algorithms to work on extensive degree sequences. To produce a graph representing such data, it can be beneficial to find a way to pick a sample as effectively and arbitrarily as possible from the space of all possible graphs that satisfy the degree sequence.

[15]Fosdick has provided some real-life applications of sampling.

- Collaboration networks can be defined as a network of people working with each other towards the same goal. An example of a collaboration network is a network of researchers collaborating on studies. In such collaboration networks, it's interesting to observe patterns in their collaboration habits. Degree assortativity is a property that measures the tendency for nodes of high degrees to be connected to other high degree nodes. In the case discussed, if the researches are the nodes and the number of collaborations per person to be the number of edges, i.e., node degree, it can be worthwhile to observe this property.
- Fosdick mentions a barn swallow interaction network, wherein it is valuable to observe if birds of similar traits interact more. Trait assortativity is a property that measures the similarity of scalar-valued traits between a pair of joined nodes in a graph. This property can be inferred from a representative graph wherein the trait can be considered the vertex label and the trait value as the vertex degrees.
- A vertex space can be pictured to be partitioned into different groups based on the patterns of the edges. Communities can be considered to be vertices that are clustered more densely than one would expect. As stated before, it is essential to be able to sample graphs for these large spaces to be able to observe and detect remarkable communities.

It is important to note that the above-mentioned network properties can be estimated from labeled graphs representing such large-sized networks. The generation of these large graphs can be made feasible through sampling techniques for picking a single random sample uniformly from the space of all labeled graphs that satisfy the degree sequence.

5.3 Modified James-Riha sampling algorithm

The James Riha exhaustive algorithm mentioned in the previous chapter has been modified to achieve sampling. This is done by introducing randomization at 2 levels:

1. As opposed to the exhaustive method where the solutions are iterated one by one, the randomized method picks a random solution from the space of all possible solutions.
2. In the exhaustive version, the nodes were picked from the equivalence classes according to their occurrence in the class, i.e., their labels. In the random method, the nodes are picked randomly from the equivalence classes.

Consider an example of generating a graph on 5 nodes with degree sequence [2,2,2,2,2]. The different equivalence classes and solutions are shown in Figure 5.1, and the different stages in the graph building are shown in Figure 5.2.

Node 1 is selected as the first node to saturate. The other nodes, i.e., 2,3,4, and 5, all fall into the same equivalence class $Eq0$ since they have the same degrees, leading to a single solution [[2]].

Nodes 2 and 5 get picked at random from the equivalence class, thereby resulting in edges (1,2) and (1,5) and a residual degree sequence [2,1,1] that is realizable, allowing us to recurse further. The hub node selected next is 3, with the rest of the nodes falling into two equivalence classes $Eq0$ (Nodes 2 and 5 of degree 1) and $Eq1$ (Node 4 of degree 2). The solutions generated are [[2,0],[1,1]]. Unlike the exhaustive generation, the choice is made randomly this time. Solution [2,0] is randomly picked, it results in a non-realizable degree sequence [2].

This then leads us to return to the most recent choice made. We then randomly choose between the other unvisited solutions at the step, which is [1,1] in this case. On choosing this solution, edges (3,4) and (3,2) get introduced with a residual degree sequence [1,1] which is realizable.

Node 4 is the final hub node, with a single equivalence class $Eq0$ and a single solution [1], leading to the edge (4,5). This terminates the process with a successfully labeled graph realizing the inputted degree sequence.

The modified James Riha sampling algorithm takes less time than the exhaustive

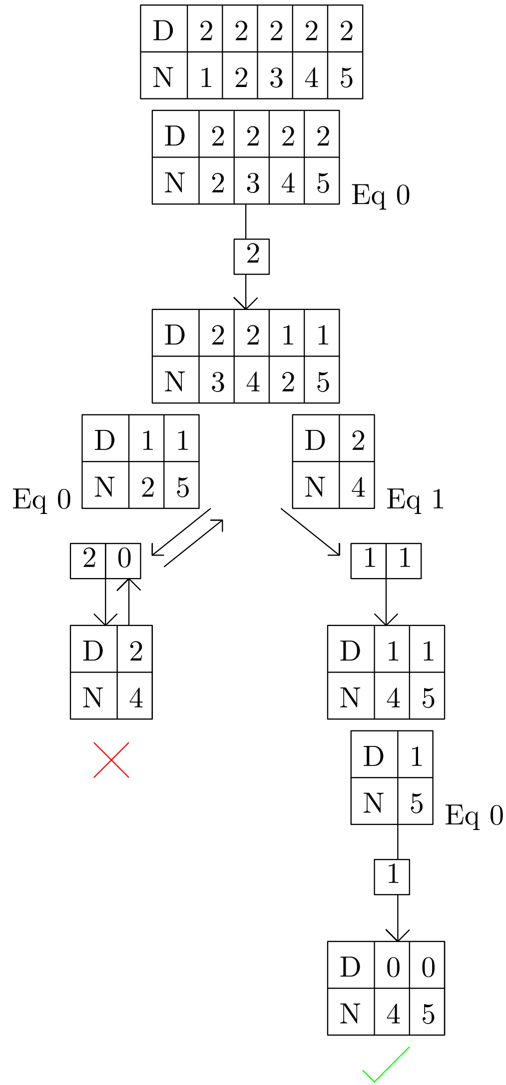


FIGURE 5.1: The different paths traversed in the James Riha sampling algorithm

version since the sampling algorithm only has to produce a single graph. The sampling algorithm would also need to traverse through lesser solutions than the exhaustive algorithm. But the algorithm falls short because of its inability to count the number of paths that might fail before generating the successful graph. This uncertainty complicates the estimation of the complexity of the algorithm.

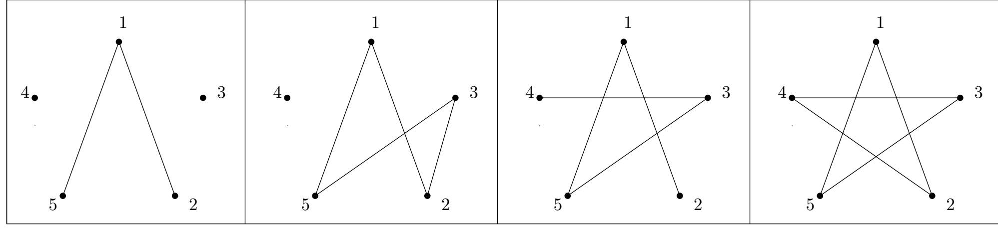


FIGURE 5.2: Graph generated by James Riha modified algorithm to make it sample

Algorithm 13: JRsampling(parameters:*Seq*)

Data: *Seq*: 2D array [[Node label, Degree],...]

Result:

1 Global variables: n- number of nodes, $G(V, E)$- Graph being generated;

2 Remove zero-degree nodes from *Seq* and sort the list according to the degrees;

// Hub will be $Seq[1]$ i.e. first element

3 **if** *All nodes are saturated* **then**

4   Display G ; Exit the recursion;

5 $eligibleNodes \leftarrow Seq[2:]$;

6 $eqc \leftarrow$ list of nodes in $eligibleNodes$ grouped by their degrees;

7 Shuffle each list in *eqc*;

8 **if** $len(eligibleNodes) <$ *Hub degree* **then**

9   **return**

10 **else**

11     $eqLen \leftarrow$ list containing lengths of each eq class;

12     $sol \leftarrow$ Call$jrsplit(Seq[0][1], len(eqc), eqLen)$; Shuffle *sol*;

13   **for** i *from* $1, \ldots, len(sol)$ **do**

14   **for** j *from* $1, \ldots, i$ **do**

15   **for** k *from* $1, \ldots, sol[i][j]$ **do**

16   Add (Hub node, $eqc[j][k]$) to E and update *Seq*;

17   **end**

18   **end**

19   Call $JRsampling(Seq)$;

20   Undo edges to move on to next solution;

21   **end**

5.4 Modified Kim's sampling method

Kim's first exhaustive algorithm has been modified to make it sample a single labeled graph instead of generating the multiplicity of labeled graphs that were previously being generated. Kim's exhaustive algorithm picks a hub node and creates the rightmost adjacent set and a set through smaller sets to generate multiple graphs.

In the modified version, we generate the rightmost adjacency set and randomly generate a single smaller set. The comparative robustness of the modified method can be attributed to two reasons, namely:

1. The modified version needs to generate a single labeled graph that satisfies the input degree sequence, whereas the exhaustive version has to generate multiple graphs.
2. In the exhaustive version, multiple smaller sets are generated for each rightmost adjacency set calculated. The modified version won't improve if a single smaller set is randomly picked from the set of all smaller sets. Instead, in our algorithm, we randomly generate the smaller set nodes, which improves overall efficiency.

The process can be better understood with an example generation of a graph on 8 nodes with degree sequence [3,3,2,2,2,2,1,1]. Figure 5.3 shows the different states of the generation.

Node 1 gets picked as the first hub node. Nodes 6, 7, and 8 forms the rightmost set. On invoking the function to generate a smaller set, [4,6,7] randomly gets generated, leading to edges (1,4), (1,6), and (1,7). We're then left with the residual degree sequence [3,2,2,1,1,1] (sorted and removed zero degree nodes).

The same process gets repeated with 2 picked as the hub node, leading to edges (2,5), (2,6), and (2,8). Finally, node 3 gets picked as the hub node, introducing the edges (3,4) and (3,5), thereby concluding the entire generation process since all the nodes get fully saturated.

The algorithms are stated below:

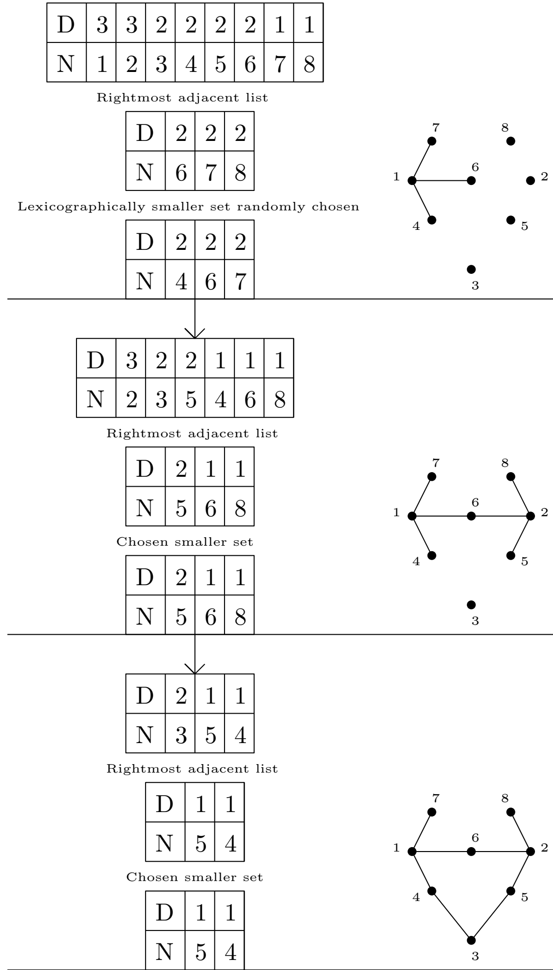


Figure 5.3: Graph generated by the modified Kim algorithm

Algorithm 14: RandomSmallSet(parameters: $NodeList$,*Rightmost*)

Data: $NodeList$: Set of node labels sorted in descending order of degrees,

Rightmost: rightmost adjacency set

Result: *SmallSet*: Randomly generated adjacency set that's $\leq$ *Rightmost*

1 *SmallSet* $\leftarrow$ [];

2 *LowerLimit* $\leftarrow$ min($n1$);

3 **for** *i from* $0, \ldots, len(Rightmost)$ **do**

4   **while** *RandomPick not in* $NodeList$ **do**

5   *RandomPick* $\leftarrow$ pick randomly from range(*LowerLimit*, $\ldots$, *Rightmost*[i]);

6   **end**

7   append *RandomPick* to *SmallSet*;

8   *LowerLimit* $\leftarrow$ *RandomPick* $+ 1$;

9 **end**

10 **return** *SmallSet*

Algorithm 15: ranKimRecGen(parameters: *Seq*)

```
   Data: Seq:2D array [[Node label, Degree],...]
   Result:
1  Global variables: G(V, E)- Graph;
2  NodeList ← nodes from Seq; Remove zero degree nodes from Seq and sort it in
     decreasing order of the degrees;
   // Hub will be Seq[1] i.e. first element
3  n ← Number of nodes;
4  if All nodes are saturated OR Seq is empty then
5  |  Display G; Exit to the main;
6  else
7  |  Rightmost ← [];
8  |  for i from n,...,1 do
9  |  |  if len(Rightmost) < Hub degree then
10 |  |  |  Temporarily add (Hub node, Seq[i][0]) to E and update Seq with
   |  |  |    temporary changes;
11 |  |  |  if C.G. Test(Seq== True then
12 |  |  |  |  Append Seq[i][0] to Rightmost
13 |  |  |  end
14 |  |  |  Remove (Hub node, Seq[i][0]) from E and update Seq;
15 |  |  end
16 |  |  else
17 |  |  |  break;
18 |  end
19 |  Sort Rightmost;
20 |  if there are unsaturated nodes left then
21 |  |  SmallerSetIndices ←Call RandomSmallSet(NodeList, Rightmost);
22 |  |  SmallSet ←Translate the indices in SmallerSetIndices into node labels;
23 |  |  for i in SmallSet do
24 |  |  |  Add (i,Hub node) to E;
25 |  |  |  Update Seq;
26 |  |  end
27 |  Call ranKimRecGen(Seq);
28 |  return
```

5.5 Kim's follow-up sampling algorithm

Kim proposed a follow-up [6] to his previous exhaustive method. The follow-up method aims to sample a single graph instead of exhaustively generating multiple graphs. For a collection of nodes ranging from 1 to n, let the non-increasing degree sequence be $d_1, d_2, \ldots, d_n$. If node 1 is the hub node, then the d_1 nodes after node 1 are considered the leftmost adjacency set. These nodes tend to preserve graphicality when connected to the hub node.

As mentioned in the previous sections, the failure starts happening beyond the boundary of the leftmost adjacency set. Kim's sampling method aims to push the boundary of the leftmost adjacency set as far to the right as possible. This is done by iterating through the nodes from right to left and making temporary connections to obtain the point at which graphicality breaks. Kim[6] states that all nodes with equal or smaller degrees (i.e., to the right) than the failure node also break graphicality.

In the original method, the failure node is obtained by iterating through the nodes from right to left and performing Erdos-Gallai tests to check if connecting the iterated nodes break graphicality. We have replaced this by running a binary search instead to locate the point at which graphicality breaks, reducing the number of graphicality tests performed.

The first connection at each hub node always maintains graphicality, allowing the first connection to be made to any node randomly picked from the other available nodes. The connections after this straightforward connection require some computation. The connections after the first connection are made by connecting the hub node to a node randomly picked from an allowed set and avoiding the nodes in the forbidden set. The method is executed recursively, with the repeated operation being the following:

1. Sort the degree sequence in non-increasing order of the degrees
2. Pick the highest degree node as the hub node, i.e., the node to saturate
3. Make the first connection to any available node and add the hub node and the chosen node to the forbidden set
4. The connections following the first connection are made to a node from an allowed set.
   (a) The nodes to the right of the leftmost adjacent set are considered the candidate nodes for the allowed set.

(b) Temporarily reduces the degree of the hub node to 1 by assuming that the hub node is connected to all the nodes from the leftmost adjacent set except for the last one.

(c) With this as the setup, a binary search is performed over the candidate nodes to locate the largest degree node that fails the graphicality test (if a failure node exists)

(d) Compute the allowed set according to the failure node if located and randomly pick a node from the allowed set to connect to the hub node.

5. The above process is repeated until the hub node is saturated. The entire process goes on till all the nodes are fully saturated.

The randomization comes from the node being picked randomly from an allowed set for the connections. This process can be better understood by going through generating a graph on 5 nodes with the degree sequence [2,2,2,2,2].

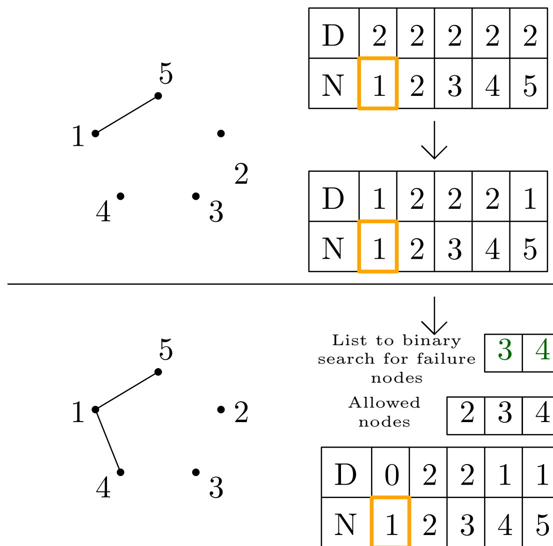


FIGURE 5.4: Kim's sampling algorithm: Saturating node 1

We start with 1 as the hub node to saturate (node 1's saturation process can be seen in Figure 5.4). The first connection never fails; node 5 is randomly picked, leading to the edge (1,5). Nodes 1 and 5 form the forbidden set. The connections succeeding the first connection involve building an allowed set in the following way. The allowed set cannot have nodes from the forbidden set. The nodes forming the leftmost adjacency set (that are not part of the forbidden set) are by default a part of the allowed set. The degree of the hub node is 1, which means that the leftmost adjacent set is [2] and gets appended to the allowed set. The remaining nodes [3,4] need to be binary searched to locate the failure node if it exists. On performing this search, we find that neither nodes

fail the test, thereby resulting in the allowed set [2,3,4]. Node 4 gets randomly picked from the allowed set, leading to the edge (1,4).

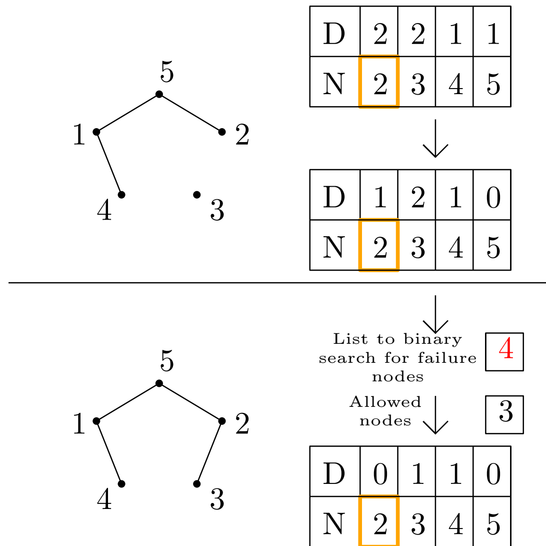


FIGURE 5.5: Kim's sampling algorithm: Saturating node 2

Figure 5.5 shows how the second hub node, i.e., node 2, gets saturated. As seen previously, the first connection is made randomly, and edge (2,5) is introduced. The allowed set already consists of nodes from the leftmost adjacent set for the second connection, i.e. [3]. The list to be binary searched is a single node [4], which fails the test, resulting in the allowed set [3]. The hub node is joined with a node picked from the allowed set, i.e., 3, leading to the edge (2,3), thereby fully saturating node 2.

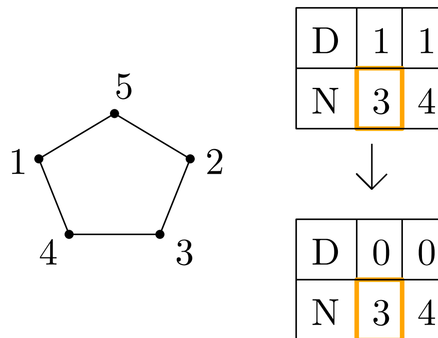


FIGURE 5.6: Kim's sampling algorithm: Saturating node 3

As shown in Figure 5.6, the final hub node, i.e., node 3, gets saturated in a single step since the degree at the hub node is 1, leading to the edge (3,4). This concludes the generation since all the nodes become fully saturated. The algorithm can be formally

stated as below:

Algorithm 16: BinarySearch(parameters: $CandidateNodes$, Seq)

Data: $CandidateNodes$: List of nodes to search for failure node, Seq: 2D array [[Node label, Degree],...]

Result: $allowedSet$: Allowed set

```
1  e ← len(Seq) − 1; start ← 0; mid ← floor((start + e)/2);
2  while ((e − start) ≥ 0 do
3      mid ← floor((start + e)/2);
4      for i from 0, ..., len(Seq) do
5          if CandidateNodes[mid] == Seq[i][0] then
6              Decrement Seq[i][1];
7          end
8      end
9      DegreeSeq ← Degrees of nodes in Seq; Sort DegreeSeq;
10     if erdosTest(DegreeSeq) then
11         start ← mid + 1; flag ← len(L);
12     end
13     else
14         e ← mid; flag ← start;
15     for i from 0, ..., len(Seq) do
16         if L[mid][0] == Seq[i][0] then
17             Increment Seq[i][1]
18         end
19     end
20     if start == e == mid then
21         break;
22     end
23 end
24 for i from 0, ..., flag do
25     append Seq[i][0] to allowedSet
26 end
27 return allowedSet
```

Algorithm 17: KimSampGen(parameters: *Seq*)

```
  Data: Seq: 2D array [[Node label, Degree],...]
  Result: G(V,E): Graph that satisfies Seq
1 choice ← 0;
2 while Seq has nodes with non-zero degrees AND Seq is not empty do
3     candidateNodes ←[]; forbiddenSet ←[]; // Hub will be Seq[1] i.e. first
        element
4     Add Hub node to forbiddenSet; Add Seq[2 :] to candidateNodes;
5     choice ← randomly pick node from candidateNodes;
6     Add(choice,Hub node) to E and update Seq;
7     Add choice to forbiddenSet and remove choice from candidateNodes;
8     while Hub node is unsaturated do
9         candidateNodes ←nodes that are not in forbiddenSet in sorted order;
10        leftmost ←[];
11        if Hub node's degree is not 1 then
12            for i from 1,...,Hub Degree) do
13                Append candidateNodes[i] to leftmost
14            end
15            Temporarily connect hub node to nodes from leftmost until Hub
                degree is 1 and update Seq;
16        end
17        else
18            Append nonfbdn[-1] to leftmost;
19        if candidateNodes is not empty then
20            allowedSet ←Call BinarySearch(candidateNodes,Seq);
21        else
22            allowedSet ←[];
23        allowedSet ← Combine leftmost and allowedSet;
24        choice ←randomly pick a node from allowedSet;
25        Add(choice,Hub node) and update Seq;
26        Add choice to forbiddenSet;
27    end
28    Sort Seq and remove nodes with degree 0;
29 end
```

5.6 Complexity Analysis

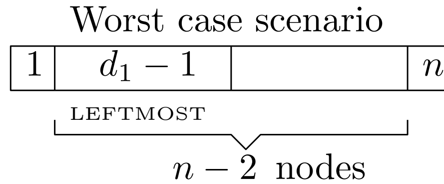
5.6.1 James Riha sampling method

The algorithm saturates nodes by constructing solutions. Each solution corresponds to how the hub degree can be derived from the nodes from the different equivalence classes. The number of such solutions is an open problem since it depends on the distribution of degrees in the input degree sequence [16]. Additionally, the failure solutions cannot be characterized, which poses inconsistencies. The irregular failure rate also makes it difficult to analyze its complexity.

5.6.2 Modified Kim-1 sampling method

Kim's sampling algorithm operates by calculating the rightmost adjacent set and generating a random smaller set. Let the number of nodes be n, and the degrees of the nodes be $d_1, d_2, \ldots, d_n$. If the degree at node 1 is d_1, then the $d_1 - 1$ nodes after node 1 constitute the leftmost adjacent set. The graphicality tests conducted have a complexity of $O(n)$.

The rightmost adjacent set is calculated by traversing the nodes from right to left and conducting graphicality tests to determine the set. The worst case that can occur is as follows:



The worst-case arises when all the nodes past the leftmost set are failure nodes,i.e., they all fail the graphicality test. This would imply that:
The number of failure nodes:
$(n-2) - (d_1 - 1)$
$= n - d_1 - 1$

Every failure node would require a graphicality test which would result in the following complexity:

$O(n - d_1 - 1)$ x n [n for the graphicality tests]
$= O(n^2)$

With n nodes in the graph, the total complexity is of $O(n^3)$ (applying the sum of cubes property)

5.6.3 Kim's second sampling method

Kim's follow-up method that samples instead of enumerating works by creating allowed sets and forbidden sets to control the way connections are made. As stated in the previous sections, computing the allowed set is expensive since it involves making numerous temporary connections and checking for graphicality.

Kim[6] claims that his second sampling algorithm can run in at most $O(n^3)$ steps. The n^3 can be accounted for in the following way:

- In the worst case, the maximum number of links in a simple graph is $O(n^2)$
- The naive approach of performing graphicality tests for all the nodes to find the failure node. This method of calculating the point of failure takes $O(n^2)$ for each connection. In the worst case with $O(n^2)$ connections, the total complexity becomes $O(n^4)$.
- The improvised method by Kim[6], proposes an optimized way of checking for calculating the failure node. The failure node connection can be detected through a recursive way of calculating the left and right sides of the Erdos-Gallai condition, which brings the complexity of the calculation of the failure node to $O(n)$. The total complexity is therefore $O(n^3)$.

On the other hand, our modification eliminates the need for graphicality tests for all the nodes in the naive approach, by using a simple binary search. The binary search modification requires a graphicality test for $O(\log n)$ nodes to locate the failure node (Binary search's complexity is $\log n$). Therefore each connection would take $O(n \log n)$ time. The total complexity would then be $O(|E|\ n \log n)$. If $|E| = O(n)$, the complexity is $O(n^2\ logn)$.

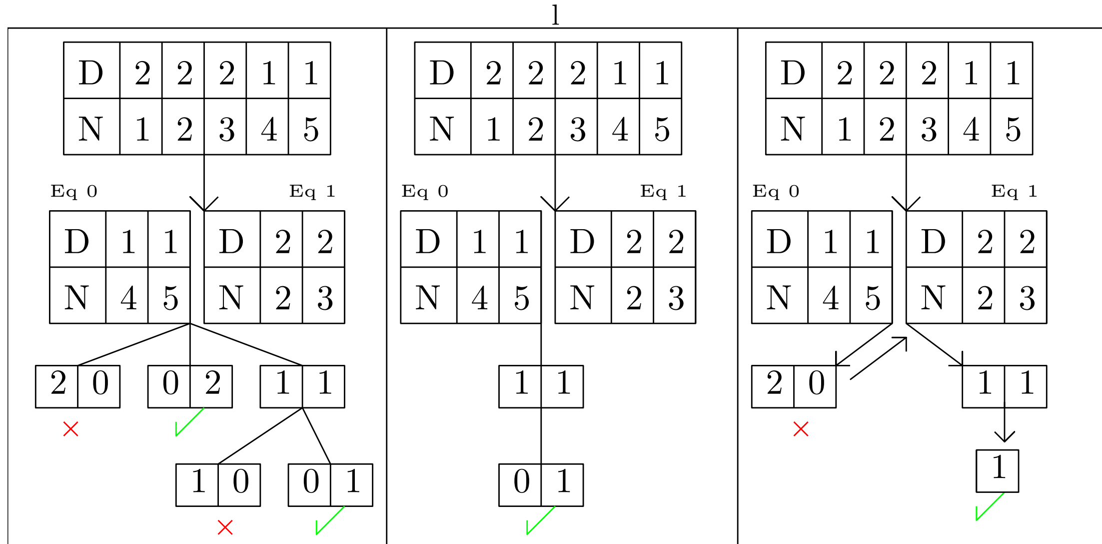


FIGURE 5.7: Side by side comparison of James Riha exhaustive method (i)James Riha exhaustive (ii)James Riha sampling advantageous run (iii)James Riha sampling wasteful run

5.7 James Riha exhaustive VS James Riha sampling

Figure 5.7 shows a side-by-side comparison of the James Riha exhaustive method and the James Riha sampling method. The example graph generation shown is for a graph on 5 nodes with the degree sequence [2,2,2,1,1].

Figure 5.7 (i) shows the exhaustive James Riha leading to a total of 4 paths, where the first success is in the second traversal. On the other hand, Figure 5.7 (ii) shows an advantageous run of the modified James Riha method, wherein success is found in the first run since the successful solution got randomly picked first.

But the randomness of this choice can lead to a delayed find of success as well. Figure 5.7 (iii) describes this situation wherein there are two backtracks because randomly picking the wrong solution twice before locating the successful path. This wasteful scenario describes the uncertain James Riha sampling method (because of the failed paths).

While James Riha is better in not having as many graphicality tests, it suffers due to the inability to predict the number of failures. This can be understood from the tabulation of the average number of failures encountered (10 trials), done for 10 different degree sequences. There is no way to characterize the fail solutions in the method, which makes it difficult to quantify the algorithm's complexity.

S.No	Number of nodes	Degree sequence	Average number of failures
1	5	2,2,2,2,2	0.2
2	5	2,2,2,1,1	0.5
3	6	3,3,3,2,2,1	0.6
4	6	3,3,3,3,3,3	0.8
5	7	2,2,2,2,2,2,2	0.48
6	7	4,4,4,4,4,2,2	1.68
7	8	3,2,2,1,1,1,1,1	0.18
8	8	3,3,2,2,2,2,1,1	0.56
9	10	6,6,5,5,4,4,3,3,2,2,1	1.14
10	16	3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3	0.56

TABLE 5.1: Recording of number of failures before a graph is generated for 10 cases (10 trials for each case)

5.8 Kim exhaustive VS Kim modified sampling

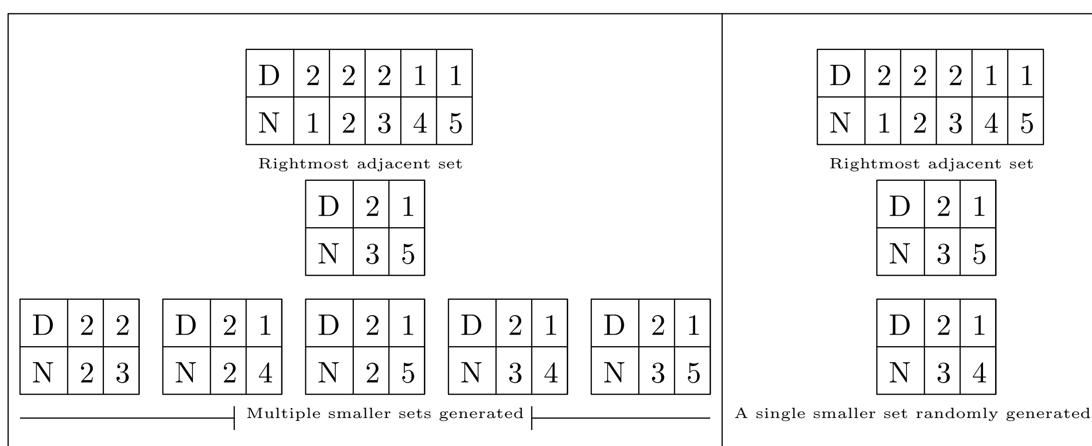


FIGURE 5.8: Side by side comparison of Kim's first method and its modification (i)Kim exhaustive generation (ii)Modification of Kim's method for sampling

Figure 5.8 is based on an example generation for a graph on 5 nodes with the degree sequence [2,2,2,1,1]. Figure 5.8 (i) shows how the exhaustive method generates five different smaller steps at every point corresponding to a calculated rightmost adjacent set. This operation is expensive both in terms of time and space. This is improved in the modified method to sample in; wherein a single smaller set is generated at every step corresponding to a rightmost adjacent set, as seen in Figure 5.8 (ii).

5.9 Comparing the modified methods: James Riha sampling vs. Kim 1 sampling

Both the algorithms were run for the same degree sequences to draw a comparison between their performance. We ran both the algorithms for a total of 10 cases, 10 times each, and calculated an average time taken of the 10 trials. The tareftab:jrrankim1rantable is the record of the values. The algorithm has been tried for graphs of relatively bigger sizes to see a concrete difference in run times.

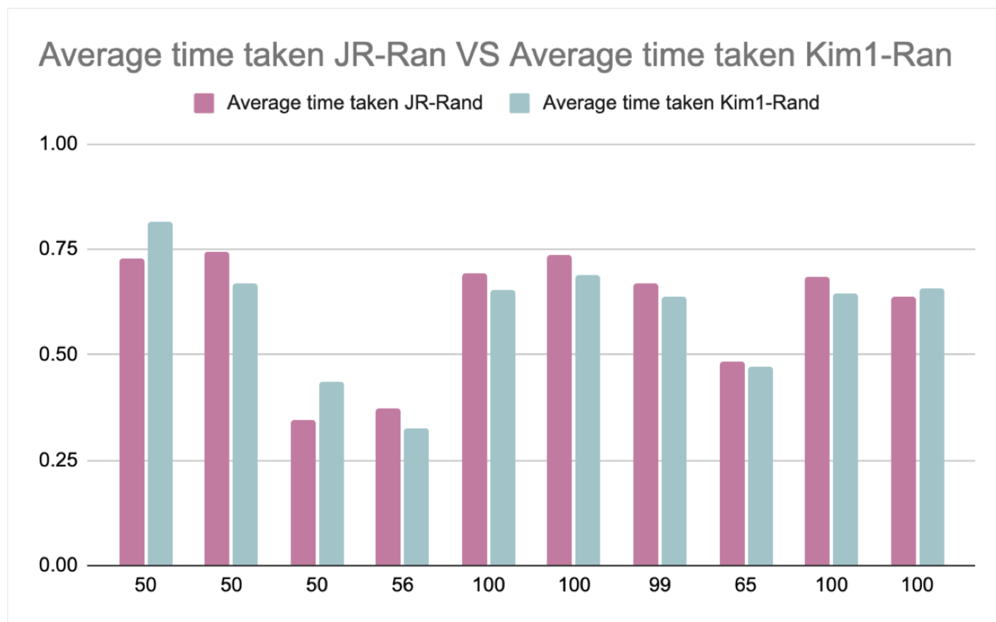


FIGURE 5.9: Graph comparing the run times of James Riha sampling and Kim 1 sampling

Figure 5.9 shows a visualization of the data in Table 5.2. The visualization compares the run times of the James Riha sampling method and the Kim 1 sampling method. The visualization shows that the James Riha method takes longer than the modified Kim 1 sampling method in most cases.

This can be attributed to the unpredictable failures faced by the James Riha method, as mentioned previously. Even though the modified Kim's first sampling method doesn't always perform better than the James Riha sampling, the former manages to be more consistent. It doesn't have the uncertainty of facing failures. This thereby makes the modified Kim's first sampling method the better one out of the two.

S.No	Nodes	Degree sequence	Average time taken JR-Rand	Average time taken Kim1-Rand
1	50	3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3, 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3	0.72907	0.81627
2	50	4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4, 4,4,4,4,4,4,4,2,2,2,2,2,2,2,2,2,2,2,2,2,2	0.74329	0.66805
3	50	4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3	0.34397	0.43822
4	56	4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	0.37483	0.32706
5	100	7, 6, 5, 4, 4, 4, 4, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	0.69311	0.65465
6	100	3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3	0.73713	0.68735
7	99	4, 4, 4, 4, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	0.66855	0.63673
8	65	6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2	0.48355	0.47295
9	100	99, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	0.68616	0.6452
10	100	5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3	0.63868	0.65601

Table 5.2: A tabulation comparing the run times of JR-Ran and Kim 1-Ran for 10 different cases (10 trials each)

Chapter 6

Concluding Thoughts

This thesis is a detailed study of the different algorithms that can generate graphs when given a degree sequence. We start by discussing some primary conditions that can be used to check for graphicality. We've also established some applications of graph generation.

This leads us to understand some classic graph generation methods like the Havel-Hakimi generation and the Tripathi generation in Chapter 1. We have added randomization to the algorithm in the Havel-Hakimi method, which stops it from consistently generating the same graph for a given degree sequence.

We touch upon some special cases of forests and split graphs and propose algorithms for generating the same.

We then explore generation methods like the James Riha method and Kim's first method, which are both exhaustive, to understand the methodologies. Once we understood the concept, we managed to modify these techniques to sample instead of exhaustive generation.

The applications and advantages of sampling are also briefly discussed. We then discuss the sampling methods in detail, go over the methodology, and simulate them with examples. A follow-up method by Kim is also discussed. We have added a modification to this method as well, the effect of which is also discussed.

The complexities of the sampling algorithms are also computed, followed by a comparison of the different methods using examples and graphs.

6.1 Future Works

The degree sequences can be characterized to understand how the generation process varies for different degree sequence distributions.

An open problem discussed in the paper is counting the different solutions and characterizing the solutions that fail in the James Riha algorithm. This would help with optimizing both the exhaustive and sampling versions of the method.

It will be worth looking into counting the number of graphs given a degree sequence. Once this is quantified, it will certainly help generate graphs given a degree sequence, efficiently and uniformly at random.

Bibliography

[1] G Tinhofer. Generating graphs uniformly at random. In *Computational graph theory*, pages 235–255. Springer, 1990.

[2] Amitabha Tripathi, Sushmita Venugopalan, and Douglas B West. A short constructive proof of the erdős–gallai characterization of graphic lists. *Discrete mathematics*, 310(4):843–844, 2010.

[3] S Louis 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.

[4] KR James and W Riha. Algorithm 28 algorithm for generating graphs of a given partition. *Computing*, 16(1-2):153–161, 1976.

[5] Hyunju Kim, Zoltán Toroczkai, Péter L Erdős, István Miklós, and László A Székely. Degree-based graph construction. *Journal of Physics A: Mathematical and Theoretical*, 42(39):392001, 2009.

[6] Hyunju Kim, Charo I Del Genio, Kevin E Bassler, and Zoltán Toroczkai. Constructing and sampling directed graphs with given degree sequences. *New Journal of Physics*, 14(2):023012, 2012.

[7] Václav Havel. A remark on the existence of finite graphs. *Casopis Pest. Mat.*, 80:477–480, 1955.

[8] Milena Mihail and Nisheeth K Vishnoi. On generating graphs with prescribed vertex degrees for complex network modeling. *Position Paper, Approx. and Randomized Algorithms for Communication Networks (ARACNE)*, 142, 2002.

[9] T. Gallai P. Erdős. Graphs with prescribed degrees of vertices (hungarian). *Mat. Lapok 11*, pages 264– 274, 1960.

[10] Martin Charles Golumbic. Chapter 6 - split graphs. In Martin Charles Golumbic, editor, *Algorithmic Graph Theory and Perfect Graphs*, pages 149–156. Academic Press, 1980.

[11] Vladislav Bína and Jiří Přibil. Note on enumeration of labeled split graphs. *Commentationes Mathematicae Universitatis Carolinae*, 56(2):133–137, 2015.

[12] Alexey S Rodionov and Hyunseung Choo. On generating random network structures: Trees. In *International Conference on Computational Science*, pages 879–887. Springer, 2003.

[13] Edward A Bender. The asymptotic number of non-negative integer matrices with given row and column sums. *Discrete Mathematics*, 10(2):217–223, 1974.

[14] Charo I Del Genio, Hyunju Kim, Zoltán Toroczkai, and Kevin E Bassler. Efficient and exact sampling of simple graphs with given arbitrary degree sequence. *PloS one*, 5(4):e10012, 2010.

[15] Bailey K Fosdick, Daniel B Larremore, Joel Nishimura, and Johan Ugander. Configuring random graph models with fixed degree sequences. *Siam Review*, 60(2):315–355, 2018.

[16] W Riha and KR James. Algorithm 29 efficient algorithms for doubly and multiply restricted partitions. *Computing*, 16(1):163–168, 1976.

Vita Auctoris

Name: Amreeth Rajan Nagarajan

Place of birth: Tamil Nadu, India

Year of birth: 1997

Education: Bachelor of Engineering in Computer Science from Birla Institute of Technology and Sciences, Dubai, U.A.E (2015-2019)

Master of Science in Computer Science from University of Windsor, Windsor, Canada (Fall 2019- Summer 2021)