FASTER ISOMER NETWORK GENERATION

by

Dheivya Thiagarajan

© Copyright by Dheivya Thiagarajan, 2017

All Rights Reserved

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Mathematical and Computer Sciences).

Golden, Colorado

Date ________________________

Signed: ________________________
Dheivya Thiagarajan

Signed: ________________________
Dr. Dinesh Mehta
Thesis Advisor

Golden, Colorado

Date ________________________

Signed: ________________________
Dr.Tracy Camp
Professor and Head
Department of Computer Science

ABSTRACT

Isomer networks provide a mechanism to understand and interpret relationships between organic molecules with applications in medicinal chemistry and drug design. The extraction of isomer networks is a time and data-intensive computation. The contributions of this dissertation are a variety of techniques to more efficiently (with respect to time and memory) compute isomers networks. Specifically, we describe our efforts to improve the network extraction process by 1) Using the symmetry present in most molecules to reduce run time and memory and streamlining the algorithm used for the detection of duplicate dnNames[1], a key step in determining the bond count distances between pairs of isomers. Together, these techniques result in reductions in memory of up to 60% and improvements in runtime of up to a factor of 100. 2) Developing an optimal grouping algorithm to subdivide an all-all computation with large memory requirements. The algorithm provides a solution to sub divide the "big data" problem that arises in the construction of isomer networks into several independent "small data" problems. Our results show that using the grouping algorithm can help divide large data sets into independent smaller ones that can be processed in parallel. 3) Generating the isomer network for 1,050,125 isomers of Nicotine (with a preliminary analysis of the same) using the cloud computing capabilities of Amazon Web Services[2] and Microsoft Azure[3]. These techniques can also be employed to successfully compute isomers networks for other chemical compounds.

TABLE OF CONTENTS

ABSTRACT . . . iii

LIST OF FIGURES . . . vii

LIST OF TABLES . . . x

ACKNOWLEDGMENTS . . . xiv

DEDICATION . . . xv

CHAPTER 1 INTRODUCTION . . . 1

CHAPTER 2 BACKGROUND . . . 5

2.1 Isomer Networks . . . 5

CHAPTER 3 IMPROVED ALL PAIR SIMULTANEOUS ALL-ALL (SAA) ALGORITHM . . . 9

3.1 Group Theory/Symmetry . . . 10

3.2 Application of Symmetry to SAA . . . 13

3.3 Implementation Details (Special Case 1) . . . 18

3.4 Preprocessing Steps . . . 18

3.5 Step 5: Improved SAA . . . 24

3.6 Analysis . . . 27

3.7 Results . . . 28

CHAPTER 4 GROUPING ISOMERS . . . 32

4.1 Two dimensional case ($d = 2$) . . . 35

4.1.1 Example . . . 35

4.1.2 Implementation Details 37

4.1.3 Proof of correctness 37

4.2 Three-dimensional example ($d = 3$) 39

4.2.1 Proof of correctness 41

4.3 Higher dimensional case (d>2) 42

4.4 Implementation 45

4.5 Deriving number of groups of isomers 47

4.6 Application 47

4.6.1 In-memory approach 50

4.6.2 File-based approach 51

4.6.2.1 Store dnNames and compute Nauty names as needed 54

4.6.2.2 Store Nauty names 55

4.6.3 Discussion 56

4.7 Results 56

4.7.1 In-memory approach 57

4.7.2 File-based approach 57

CHAPTER 5 ISOMER NETWORK GENERATION AND GRAPH ANALYTICS . . 61

5.1 Generating the isomer network 61

5.1.1 Genarating and storing Nauty names for each isomer 61

5.1.2 Experience and Results with AWS 63

5.1.3 Generating isomer network from Nauty names 64

5.1.4 Experience and Results with MS Azure 67

5.2 Analytics for graph with around 500K vertices 68

5.2.1 Centrality . . . 69
5.2.2 Centralization . . . 74
5.2.3 Network cohesion . . . 74
5.2.4 Community detection . . . 74
5.3 Analytics for graph with a million vertices . . . 76
5.3.1 Centrality . . . 76
5.3.2 Centralization . . . 82
5.3.3 Network cohesion . . . 82
5.3.4 Community detection . . . 82
5.4 Graph Analytics with an example graph . . . 84
5.4.1 Other metrics . . . 87
5.4.2 Centrality . . . 89
5.4.3 Centralization . . . 96
5.4.4 Network cohesion . . . 98
5.4.5 Community detection . . . 102
CHAPTER 6 CONCLUSION . . . 113
6.1 Summary . . . 113
6.2 Future directions . . . 114
REFERENCES CITED . . . 115

LIST OF FIGURES

Figure 2.1 Two C_2H_6O Isomers . . . 6

Figure 2.2 A simple isomer network for the isomers shown in Figure 2.1 . . . 6

Figure 2.3 An order-4 isomer social network for the complete isomer network of Table 2.1. All edge weights (not shown in the figure) are 4. . . . 8

Figure 3.1 Automorphisms of C_2H_6O depicted in Figure 2.1(a) . . . 10

Figure 3.2 Part of depth-first refinement tree generated by Bliss for graph shown on left . . . 13

Figure 3.3 Example Graph . . . 14

Figure 3.4 Two examples . . . 15

Figure 3.5 Cutting one bond per isomer. In (a) we cut 28 bonds per isomer (including the symmetric ones, shown in the same color), and in (b), by incorporating the concept of not cutting more than one of a set of symmetric bonds, we cut only 23 bonds per isomer. Bonds in gray are not cut. . . . 19

Figure 3.6 Cutting two bonds per isomer. In (a), we cut 378 bonds per isomer (including symmetric ones, shown in the same color), while in (b), by incorporating the concept of not cutting more than one of a set of symmetric bonds, we reduce the number of bonds cut to 257. . . . 20

Figure 3.7 Flow chart depicting the generation of isomer networks using symmetry . 21

Figure 3.8 One Nicotine ($C_{10}H_{15}N_2$) isomer represented as a chemical graph . . . 22

Figure 3.9 Flow chart describing Step 3 . . . 22

Figure 3.10 Atoms identified as symmetric following the output from Bliss . . . 23

Figure 3.11 Symmetric edges identified, from symmetric vertices. . . . 24

Figure 3.12 Flow chart describing Step 5 - new/modified steps in gray . . . 25

Figure 4.1 All Nauty names for nine isomers in memory 33

Figure 4.2 Four of 12 iterations using the grouping technique 33

Figure 4.3 Matrix of isomers 35

Figure 4.4 2D algorithm 35

Figure 4.5 Matrices before and after the first circular shift 36

Figure 4.6 Matrices before and after the second circular shift 36

Figure 4.7 Original $3 \times 3 \times 3$ matrix 39

Figure 4.8 Matrix after the first data movement 40

Figure 4.9 Matrix after the second data movement 40

Figure 4.10 Rows 0, 1, and 2 form individual C_0, C_1, and C_2 41

Figure 4.11 Steps to list isomer numbers in groups of N 47

Figure 4.12 Process processColumns() 48

Figure 4.13 Grouping isomers 48

Figure 4.14 List of steps executed by SAA algorithm 49

Figure 4.15 Flow chart for in-memory approach 51

Figure 4.16 Flow chart for file-based approach 52

Figure 4.17 Degree Neighborhood Canonical Labeling 53

Figure 5.1 Iterations using the grouping technique 65

Figure 5.2 Frequency of degree of vertices in G_{500K} 70

Figure 5.3 Frequency of cores for G_{500K} 75

Figure 5.4 Frequency of degree of vertices in G_{1M} 77

Figure 5.5 Frequency of cores 83

Figure 5.6 Graph showing fifteenth-century Florentine marriages 85

Figure 5.7 Example graph (g) 86

Figure 5.8 Graph showing k-core decomposition of g 100

Figure 5.9 Edge betweenness clustering on g 103

Figure 5.10 Fast greedy clustering on g 105

Figure 5.11 Walktrap clustering on g 106

Figure 5.12 Leading eigenvector clustering on g 107

Figure 5.13 Label propagation clustering on g 108

Figure 5.14 Infomap clustering on g 109

Figure 5.15 Louvain clustering on g 111

Figure 5.16 Optimal clustering on g 112

LIST OF TABLES

Table 1.1 Isomer network generation runtimes . . . 2

Table 2.1 Complete Isomer Network . . . 7

Table 3.1 Sample data while cutting up to 4 bonds per isomer without symmetry . . 10

Table 3.2 Permutations (in cyclic notation) showing automorphisms of C_2H_6O depicted in Figure 2.1(a) . . . 11

Table 3.3 Output from Bliss to graph shown in Figure 3.3 (labels added by us to help identify generators in the examples that follow) . . . 16

Table 3.4 Sample data indicating symmetry and priority values set for bonds in the isomer file . . . 26

Table 3.5 Results summary when cutting 1 bond in 500 isomers . . . 29

Table 3.6 Results summary when cutting up to 2 bonds in 500 isomers . . . 30

Table 3.7 Results summary when cutting up to 3 bonds in 75 isomers . . . 30

Table 3.8 Results summary when cutting up to 4 bonds in 25 isomers . . . 30

Table 3.9 Improvement factor in terms of time and space . . . 31

Table 4.1 All-pairs table (matrix is symmetric, so lower triangle is not shown) showing the algorithmic step where each pair of isomers meets. . . . 37

Table 4.2 27 elements divided into 3 rows and 3 columns. . . . 39

Table 4.3 Data (N^d elements) stored as a one-dimensional array ($mainArray$) . . . 42

Table 4.4 N^d elements of Table 4.3 divided into N rows and N columns. . . . 42

Table 4.5 Data (243 elements) stored as a one-dimensional array ($mainArray$) . . . 43

Table 4.6 243 elements of Table 4.5 divided into 3 rows and 3 columns. . . . 43

Table 4.7 Concept of circular shift . . . 44

Table 4.8 Data stored as a one-dimensional array (C_0) 45

Table 4.9 81 elements in Table 4.8 further divided into 3 rows and 3 columns. 45

Table 4.10 27 elements of column C_{00} in Table 4.9 further divided into 3 rows and 3 columns. 46

Table 4.11 Example of circular shift 46

Table 4.12 Results for Nicotine isomer 57

Table 4.13 Results for Phenmetrazine isomer 57

Table 4.14 Results for Tyrosine isomer 58

Table 4.15 Time and space required to write to file in Step 1 (dnNames) 58

Table 4.16 Time and space required to process data from file in Step 2 (dnNames) . . 59

Table 4.17 Approach 2b (Nauty names) 59

Table 5.1 Top 25 vertices of G_{500K} sorted on vertex centrality 69

Table 5.2 Top 25 vertices of G_{500K} sorted on estimated closeness centrality with cutoff = 10 71

Table 5.3 Top 25 vertices of G_{500K} sorted on estimated betweenness centrality with cutoff = 10 72

Table 5.4 Top 25 vertices of G_{500K} sorted on eigenvector centrality 73

Table 5.5 Centralization measures computed for G_{500K} 74

Table 5.6 Number of components and their sizes in G_{500K} 75

Table 5.7 Results of community detection algorithms run on G_{500K} 76

Table 5.8 Top 25 vertices of G_{1M} sorted on vertex centrality 78

Table 5.9 Top 25 vertices of G_{1M} sorted on estimated closeness centrality with cutoff = 12. 79

Table 5.10 Top 25 vertices of G_{1M} sorted on estimated betweenness centrality with cutoff = 12. 80

Table 5.11 Top 25 vertices of G_{1M} sorted on eigenvector centrality 81
Table 5.12 Centralization measures computed for G_{1M} 82
Table 5.13 Number of components and their sizes in G_{1M} 83
Table 5.14 Results of community detection algorithms run on G_{1M} 84
Table 5.15 Shortest distance between pair of vertices in g 87
Table 5.16 Eccentricity (E) of vertices in g 88
Table 5.17 Degree of vertices of g. 89
Table 5.18 Frequency of degrees in g 90
Table 5.19 Closeness centrality (C) of vertices in g 90
Table 5.20 Estimated closeness centrality (C_e) of vertices in g with cutoff = 3. 91
Table 5.21 Betweenness centrality (B) of vertices in g 92
Table 5.22 Estimated betweenness centrality (B_e) of vertices in g with cutoff = 2. . . 93
Table 5.23 Eigenvector centrality (E_c) of vertices in g 95
Table 5.24 Top 7 vertices of g sorted on vertex centrality 96
Table 5.25 Top 7 vertices of g sorted on closeness centrality 96
Table 5.26 Top 7 vertices sorted of g on betweenness centrality 97
Table 5.27 Top 7 vertices sorted of g on eigenvector centrality 97
Table 5.28 Centralization measures computed for g 98
Table 5.29 k-core vertices of g 99
Table 5.30 Component of the graph (g) each vertex belongs to. 101
Table 5.31 Edge list of individual components of g. 101
Table 5.32 Number of components and their sizes in g 101
Table 5.33 Communities returned by Edge betweenness clustering algorithm 104

Table 5.34 Communities returned by Fast greedy clustering algorithm 104

Table 5.35 Communities returned by Walktrap clustering algorithm 106

Table 5.36 Communities returned by Leading eigenvector clustering algorithm . . . 107

Table 5.37 Communities returned by Label propagation clustering algorithm 109

Table 5.38 Communities returned by Infomap clustering algorithm 110

Table 5.39 Communities returned by Louvain clustering algorithm 110

Table 5.40 Communities returned by Optimal clustering algorithm 111

Table 5.41 Result of community detection algorithms run on g 112

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor Dr. Dinesh Mehta, for his encouragement, guidance, patience, and support throughout my Ph.D. journey. His immense knowledge and advice at every step enabled me to develop an understanding of the subject and complete my dissertation.

I thank my committee members Dr. Qi Han, Dr. Bo Wu, and Dr. Cristian Ciobanu for providing me with valuable suggestions for improving my dissertation, Dr. Tracy Camp and Dr. Cyndi Rader for their encouragement, Lori Sisneros (Graduate Program Manager, CECS) and Martha Sanchez-Hayre (Program Administrator, CECS) for their willingness to help with a smile, and my fellow CS grad students for their feedback, cooperation and of course friendship.

I would like to thank Jason Timmes (Nasdaq) for his help and feedback while writing the AWS proposal.

Words cannot express my heartfelt gratitude to my mother Janaki, my father Thiagarajan, and my brother Dr. Thiagarajan Ramakrishnan (Ram). Their unstinting support and encouragement made it possible for me to attain my Ph. D. I also thank my uncles and the rest of my family for their support and encouragement.

I want to acknowledge my grandparents who have always taken pride in me. Today I have no doubt they are watching me from heaven with tears in their eyes and a smile in their face. Thank you thatha and paatti.

This day would not have come for me without the support and encouragement of my husband Parasuram. Thank you for all the sacrifices you have made, so I could fulfill my dream. I may not have always said it, but you have been my rock through this journey!!

To my parents - Janaki and Thiagarajan,

For believing in me more than I believed in myself

You taught me to never ever give up on my dreams.

To my sons - Pradhyum and Madhav,

You light up my life.

I hope I can be the mother to you that my mother is to me.

CHAPTER 1
INTRODUCTION

The *isomer network* was recently proposed by our group in collaboration with Dr. Jean-Louis Reymond's group at the University of Berne[1]. Prof Reymond leads the "Chemical Space Project"[4] at the University of Berne. The stated goal of the project is to enumerate all possible organic molecules that could be used in medicinal chemistry. This number has been estimated at 10^{60} overall ($10^{20} - 10^{24}$ for all molecules with up to 30 atoms). Techniques that are a combination of graph theory and chemical knowledge have been used to generate 166.4 billion molecules of up to 17 atoms of C, N, O, S and halogens, resulting in GDB-17 [5], the largest such database to date. The GDB almost exclusively (>99.99%) consists of new molecules and represents a vast reservoir of opportunities for drug discovery [4]. Other possible applications of this technology include flavors and fragrances chemistry.

Our group's previous contribution to the effort has been to propose that relationships between chemicals can be explicitly represented by a weighted network based on their bond count distances (BCD). The BCD between a pair of chemicals is the minimum number of bonds that must be broken and formed to convert one chemical into another. As an example of how this network structure might be used, one could extract the nearest-neighbors of a chemical by identifying chemicals in its network neighborhood, potentially resulting in very close analogs of a molecule. Our initial efforts in this direction focused on the creation of networks of isomers - chemicals with identical sets of atoms, but different structures. Isomer networks based on BCDs were found to be correlated with Tanimoto distances between the extended connectivity fingerprints (ECFPs) of the corresponding isomers [1], suggesting that isomer networks can be used to understand and interpret relationships between organic molecules. Ultimately, this is an outcome of the well-known **S**imilar **S**tructure - **S**imilar **P**roperty principle in cheminformatics; i.e., that structurally similar molecules are expected

to exhibit similar chemical or biological properties. Beyond these exploratory studies, however, we anticipate that a network structure of chemicals will bring into play a wealth of network analytics tools that were originally developed in the social network analytics community [6–11], but more recently have been utilized in Computer Science (e.g., the PageRank algorithm used by Google to rank pages based on importance in the web graph) and other areas.

However, the computation of isomer networks has proven to be extremely expensive with respect to both computational time and memory. We briefly describe prior progress in this area below:

A straightforward algorithm for computing isomer networks consists of individually computing the bond count distance between each pair of isomers in the network. However, the sub-problem of computing the bond count distance between a single pair of isomers is itself NP-complete, making the problem of computing the bond count distance between *all* pairs harder by a factor that is quadratic in the number of isomers.

A necessary strategy to address this has been to restrict the problem to one of computing k-isomer networks (rather than complete isomer networks). In addition, an improved algorithm (called the All Pairs Simultaneous All-All algorithm) resulted in a 15x improvement in runtime over the naive approach described above for an isomer network consisting of 20 Nicotine isomers. Using a combination of these two strategies we have in previous work [1] been able to obtain the computation times shown in Table 1.1 below.

Table 1.1: Isomer network generation runtimes

Isomer Set	Theoretical # of Isomers	Sample Size	Type of isomer network (k)	Runtime (in hrs)
Nicotine w/o small rings	219,490	2,934	10	59
Nicotine w/ small rings	507,964	10,000	8	101
Phenmetrazine	10,213,951	10,000	8	66
Tyrosine	7,682,352	10,000	8	22

Even though this is an improvement, this approach is clearly not scalable to the complete set of isomers. For example, the runtime to compute the 10-isomer network for 2,934 Nicotine (without small rings) isomers is approximately 59 hours (first row of table). An (oversimplified) linear extrapolation to the entire set of 219,490 isomers suggests a runtime of 4,413 hours (about half a year). The actual time complexity is super-linear, suggesting a significantly higher runtime. In addition, we project the computation to require 1.41 TB of intermediate storage for the set of 219,490 isomers of Nicotine.

The broad goals of this dissertation are therefore to employ a variety of techniques to more efficiently (with respect to time and memory) compute isomer networks. Specifically, after providing the necessary background information in Chapter 2, our contribution is as follows.

1. In Chapter 3, we describe our efforts to improve the network extraction process (All Pairs SAA) by using the symmetry present in most molecules to reduce runtime and memory and streamlining the algorithm used for the detection of duplicate dnNames[1], a key step in determining the bond count distances between pairs of isomers. Together, these techniques result in reductions in memory of up to 60% and improvements in runtime of up to a factor of 100.

2. In Chapter 4, we present an algorithm that enables fragments of the isomer network to be developed in groups that can be accommodated within the available memory resources, such that every isomer is in a group with every other isomer once and only once. The algorithm provides a solution to sub divide the "big data" problem that arises in the construction of isomer networks into several independent "small data" problems. Our experimental results show that using the grouping technique described in this chapter, can help divide large data sets into independent smaller ones that can be processed in parallel.

3. In Chapter 5, we present the network of **1,050,125** isomers of Nicotine. We describe the process of isomer network generation using the cloud computing capabilities of Amazon Web Services[2] and Microsoft Azure[3]. We also present graph analytical metrics that can be used to analyze such a network and our results from analyzing a network of approximately 500,000 and 1,000,000 isomers using the metrics described.

Finally, Chapter 6 describes future directions this research can take.

CHAPTER 2

BACKGROUND

In this chapter, we present necessary background information for the remainder of the dissertation.

2.1 Isomer Networks

Definition A *graph* is an ordered pair $G = (V, E)$, where V is a set of vertices = {1, 2, 3, $\cdots$, n} and E is a set of edges (2-element subsets of V). A simple graph is an unweighted, undirected graph containing no vertex loops or multiple edges. A *multigraph* is a graph which can have multiple edges between a pair of vertices.

A *coloring* of V is a surjective function Π from V onto $\{1, 2, \cdots, k\}$ for some k. The number of colors i.e., k is denoted by $|\Pi|$. A cell of Π is a set of vertices with some given color [12]. A pair (G, Π), where Π is a coloring of G, is called a *colored graph*.

Definition *Chemical graphs* are chemical compounds represented as colored multigraphs, where vertices and edges represent chemical atoms, and bonds respectively. Different chemical atoms are assigned different colors. Thus, $|\Pi|$ is the number of different atoms in the compound the graph represents. For example, a chemical graph of the compound C_2H_6O would have $V = \{C_1, C_2, H_1, H_2, H_3, H_4, H_5, H_6, \mathrm{O}\}$ and $|\Pi| = 3$, one color each for C, H, and O.

Definition Let $V(G)$ be the vertex set of a graph G and $E(G)$ its edge set. Then a *graph isomorphism* from a graph G to a graph H is a bijection $f : V(G) \to V(H)$ such that two vertices $uv \in$ E(G) iff $f(u)f(v) \in E(H)$.

Graph isomorphism is said to be *color preserving*, if $\Pi(v) = \Pi(\mathrm{f(v)})$ for all $v \in V$.

Definition A *canonical naming* l is a function over all graphs, mapping a graph G to a canonical name $l(G)$ such that for any two graphs G_1 and G_2, G_1 is isomorphic to G_2 iff $l(G_1) = l(G_2)$.

If canonical names can be found for two graphs, the graphs can be checked for isomorphism by simply comparing their canonical names.

Definition Two molecules are *isomers* if they have the same formula (i.e., composition of atoms), but different structures. Figure 2.1 shows an an example of two C_2H_6O isomers.

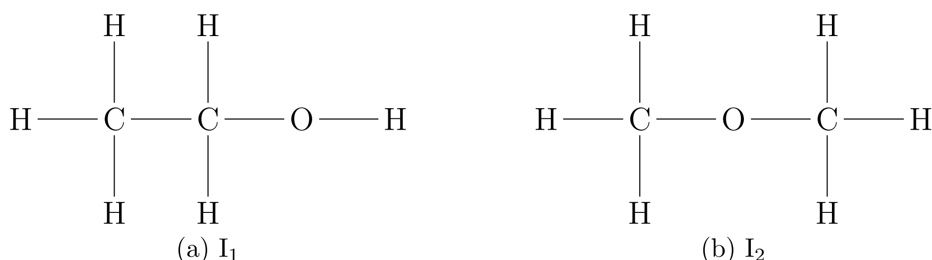


Figure 2.1: Two C_2H_6O Isomers

Definition The *Bond Count Distance* $BCD(I_1, I_2)$ between two isomers I_1 and I_2 is the *minimum* number of bonds that must be broken or formed to transform one isomer into the other. $BCD(I_1, I_2)$ in Figure 2.1 is four. This is obtained by breaking the C-C and O-H bonds in I_1 and then forming O-C and C-H bonds to get I_2. Figure 2.2 shows the resulting isomer network [1].

Figure 2.2: A simple isomer network for the isomers shown in Figure 2.1

Definition A *complete isomer network*[1] defined on a set of M isomers is a complete weighted graph on M vertices (each corresponding to an isomer) with edge weights equal to the bond count distances between the corresponding isomers.

Definition A subgraph of the complete isomer network that is limited to **edges** with BCD $\leq$ c, is a *c-isomer network*.

Definition Another subgraph of the complete isomer network that is generated by breaking up to B **bonds** per isomer is called the *reduced B-isomer network*. This is especially useful when we do not have the resources to cut all bonds in each isomer. A reduced $c/2$-isomer network serves as an approximation to a c-isomer network.

Table 2.1 shows an isomer network for 10 Nicotine isomers where entry (i, j) corresponds to $BCD(i, j)$. The matrix is symmetric, and the lower part of the triangle has been omitted for clarity. Figure 2.3 shows the 4-isomer network for the complete isomer network of Table 2.1.

Table 2.1: Complete isomer network for 10 Nicotine isomers without small rings.

		Isomer Number									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Isomer Number	**1**	0	4	4	6	4	6	6	6	4	8
	2		0	6	4	6	4	8	8	8	6
	3			0	4	4	6	8	4	8	10
	4				0	6	4	10	8	10	8
	5					0	4	4	8	8	8
	6						0	8	10	10	4
	7							0	4	4	4
	8								0	4	6
	9									0	6
	10										0

Isomer networks capture the similarity between isomers. The smaller the BCD between two isomers (as captured in the isomer network), the more similar their properties.

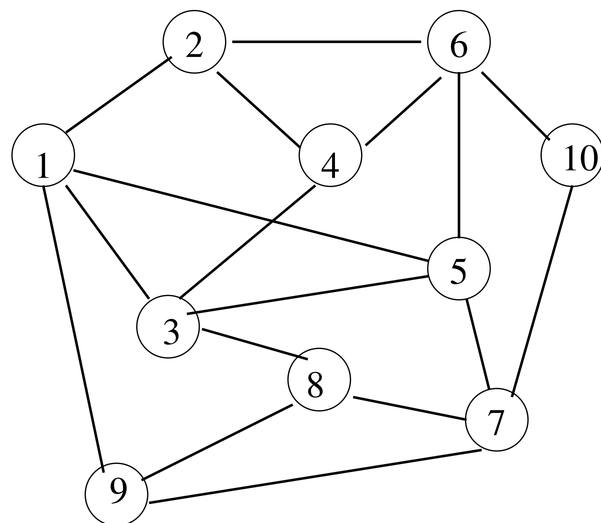


Figure 2.3: An order-4 isomer social network for the complete isomer network of Table 2.1. All edge weights (not shown in the figure) are 4.

CHAPTER 3

IMPROVED ALL PAIR SIMULTANEOUS ALL-ALL (SAA) ALGORITHM

In this chapter we discuss the improvements made to the Simultaneous AA [1] algorithm used to generate isomer networks.

The SAA algorithm [1] is used to construct a reduced B-isomer network over M isomers. It utilizes two canonical labeling algorithms: dnName and `Nauty`[12, 13]. dnName is fast, but not guaranteed to be accurate (i.e., two different chemical graphs could get the same label with a small probability) while `Nauty`[12, 13] is slower, but guaranteed to be accurate. To check whether two chemical graphs are the same, it first compares their dnNames. If they match, it then compares their `Nauty` names for confirmation.

A *reduced B-isomer network* is computed as follows. For each isomer in the set, we cut all combinations of up to B bonds. Each combination of bonds cut yields a chemical graph that may not always be connected. If the chemical graph obtained by cutting i bonds from isomer I_x is identical to the chemical graph obtained by cutting j bonds from isomer I_y, then $BCD(I_x, I_y) \leq i + j$ because it is possible to transform I_x to I_y through a set of i bond deletions and j bond additions. This approach quickly runs into space constraints, as can be inferred from data shown in Table 3.1 [1] which shows the projected space required when breaking up to 4 bonds per isomer ($B = 4$) with the existing SAA algorithm for samples of Nicotine, Tyrosine, and Phenmetrazine of sizes 2934, 10,000, and 10,000 respectively.

Extrapolating data in Table 3.1 for Tyrosine, we would need >30 TB to break 4 bonds for the 10,213,951 isomers. We propose to reduce the space required by taking advantage of the symmetry inherent in most molecules. Specifically, if two bonds are symmetric, cutting one yields the same chemical graph as cutting the other. A careful exploitation of this observation will result in improved run times and space requirements of the SAA algorithm.

Table 3.1: Sample data while cutting up to 4 bonds per Isomer

Isomer Data Sets	**Total # of isomers**	**Sample Size**	**# of atoms per Isomer**	**# of bonds per Isomer**	**Projected Space required to break upto 4 bonds in sample**
Nicotine isomers	219,490	2934	27	28	23.5 GB
Tyrosine	10,213,951	10,000	24	24	32.4 GB
Phenmetrazine	7,682,352	10,000	29	30	98.8 GB

3.1 Group Theory/Symmetry

An automorphism is an isomorphism of a graph onto itself. More formally,

Definition An *automorphism* (symmetry) of graph G is a permutation of its vertices that preserves its edge relationships. [14]

Figure 3.1 and Table 3.2 together show some automorphisms in the molecule C_2H_6O depicted in Figure 2.1(a).

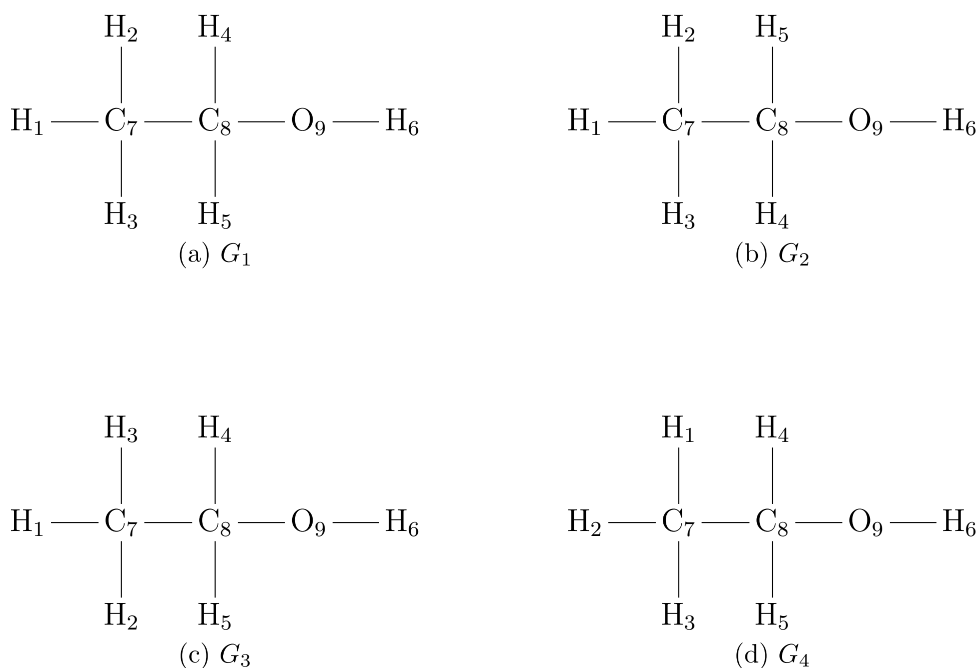


Figure 3.1: Automorphisms of C_2H_6O depicted in Figure 2.1(a)

Table 3.2: Permutations (in cyclic notation) showing automorphisms of C_2H_6O depicted in Figure 2.1(a)

Transformation	Permutation	Explanation
$G_1 \to G_2$	(1)(2)(3)(4 5)(6)(7)(8)(9)	Switch vertices 4 and 5. All other vertices stay the same.
$G_1 \to G_3$	(1)(2 3)(4)(5)(6)(7)(8)(9)	Switch vertices 2 and 3.
$G_1 \to G_4$	(1 2)(3)(4)(5)(6)(7)(8)(9)	Switch vertices 1 and 2.
$G_2 \to G_3$	(1)(2 3)(4)(5)(6)(7)(8)(9)	Switch vertices 2 and 3.
$G_2 \to G_4$	(1)(2)(3)(4 5)(6)(7)(8)(9)	Switch vertices 4 and 5.
$G_3 \to G_4$	(1 2 3) (4)(5)(6)(7)(8)(9)	Map 1 to 2, 2 to 3, and 3 to 1.

.

The set of permutations (symmetries) of G forms a group under composition (of permutations). This group is called the symmetry group of G, and is denoted by Aut (G). There are several software systems that compute symmetries [12, 13, 15–22].

Nauty was the first canonical labeling tool developed in the early 80s. The nauty search tree is based on a branching and backtracking framework, which is optimized by integrating group-theoretical techniques. Other tools such as Bliss[15–17] and saucy[18, 19] follow nauty's algorithms, but are designed to address possible shortcomings of nauty's search tree. In particular, Bliss improves the handling of large and sparse graphs, primarily in the design of data structures and subroutines accommodating large graphs and facilitating fast searching. This makes Bliss more suitable for our purposes than nauty. Although saucy is reported to be faster, we found this to be true for molecules with fewer bonds, and compounds of a single atom i.e., undirected graphs with a single color. Since we are representing chemical isomers as graphs, our graphs usually have 3 or 4 colors. Even though Bliss is primarily a graph canonicalization tool, and finding automorphisms is more a byproduct, we found Bliss to be more adaptable for our purposes.

Bliss uses the concepts of generators (defined below) and partition refinement. Also included is a brief description of how Bliss works. For a more detailed explanation, readers can refer to References [12, 14, 15], that we have used to for the following explanation.

A *generating set* of Aut(G) is a subset of the symmetries of Aut(G) whose compositions generate Aut(G). Each element of this set is a *generator*. The typical generator in Bliss represents a permutation that switches a pair of vertices with each other.

A key concept used in symmetry-detection and canonical labeling algorithms is *partition refinement*, which we describe next. An ordered partition of V is a subdivision of V into an ordered list of pairwise disjoint sets: $\Pi = (W_1, W_2, \cdots W_m)$. Each set W_i is called a cell of the partition and all vertices in the same cell are considered to have the same color. (This definition of color as used by Bliss extends the definition of color defined earlier.) A partition is *discrete* if all its cells are singleton sets and *unit* if it has only one cell (the set V). An *equitable partition* is one where every two vertices of the same color are adjacent to the same number of vertices of each color. The *refinement* of a partition is the process of subdividing one of its cells into smaller cells. Given a partition (coloring) Π, there is a unique equitable partition that is a refinement of Π and has the least number of colors. Partition refinement propagates the graph's vertex degrees and colors until the partition becomes discrete. The process of partition refinement is depicted using a tree, whose root is a unit ordered partition. The depth-first refinement process starts by choosing a non-singleton cell and individualizing each vertex in that cell. This results in refined partitions, each of which corresponds to a child of the tree node. This is repeated until the partition is discrete (resulting in a leaf node). Techniques such as coset pruning and orbit pruning are used to discard subtrees that would lead to redundant generators, making the algorithm efficient.

A *certificate* is a function that assigns a certain value to an ordered partition. Given an equitable partition (returned by partition refinement), Bliss first makes a list of edges that connect singleton cells to other cells of the partition. Then, Bliss generates the certificate by renaming each vertex in the list of edges with the index of that vertex in the partition. The vertices whose refinement lead to identical certificates are generators (and are symmetric to each other). Symmetry is also found if another node during the search produces the same certificate as the first leaf node.

We now consider an example.

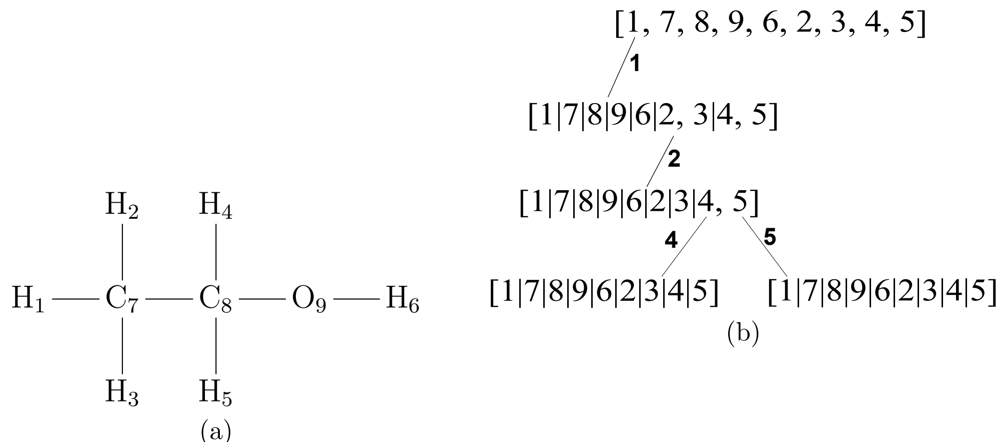


Figure 3.2: **Part** of depth-first refinement tree generated by Bliss for graph shown on left

For the graph in Figure 3.1(a) (reproduced in Figure 3.2(a)), a portion of the refinement tree generated by performing a depth-first traversal to identify equitable partitions resulting from vertex 1 is shown in Figure 3.2(b). For example, we start with vertex 1 and try to individualize each vertex. We arrive at the second level of the tree by performing a depth-first refinement from vertex 1. Vertices 2 and 3 are of the same depth and color with respect to vertex 1 and hence are in the same cell. The same applies to vertices 4 and 5. Vertices 7, 8, 9 and 6 are singleton cells since they are of different depths and colors with respect to vertex 1. We continue refining the vertices in non-singleton cells until we arrive at the leaf node, which contains only singleton cells. The certificates (leaf nodes) arrived at by refining vertices 4 and 5 are the same, indicating that (4, 5) is a generator. Note that only part of the depth-first refinement tree generated by `Bliss` is shown in Figure 3.2(b).

The next sections describe how we use generators identified by `Bliss` in our algorithm.

3.2 Application of Symmetry to SAA

Our improvements to SAA require the identification of symmetric *bonds*. Clearly, double and triple bonds can be interpreted as symmetric single bonds. Identifying additional symmetric bonds requires further processing of the symmetric vertices identified by `Bliss`. We

derive symmetric edges (bonds) from the symmetric vertices (chemical atoms) returned by `Bliss`. Consider two bonds b_1 and b_2 where $b_1 = (u_1, v_1)$ and $b_2 = (u_2, v_2)$. Now b_1 and b_2 are symmetric if $u_1 = u_2$ or u_1 and u_2 are symmetric *and* $v_1 = v_2$ or v_1 and v_2 are symmetric. (u_1, v_1) and (u_2, v_2) can also be symmetric if $u_1 = v_2$ or u_1 and v_2 are symmetric *and* $v_1 = u_2$ or v_1 and u_2 are symmetric. Without loss of generality, we henceforth consider only the first case.

We also distinguish between the cases where the number of bonds to be broken $B = 1$ versus the case when $B > 1$. When $B = 1$, all symmetric bonds identified by `Bliss` can be trivially exploited in an improved SAA algorithm. However, when $B > 1$, not all of the original bond symmetries will hold in the new graph(s) obtained by cutting one bond. We illustrate these subtleties with the graph in Figure 3.3.

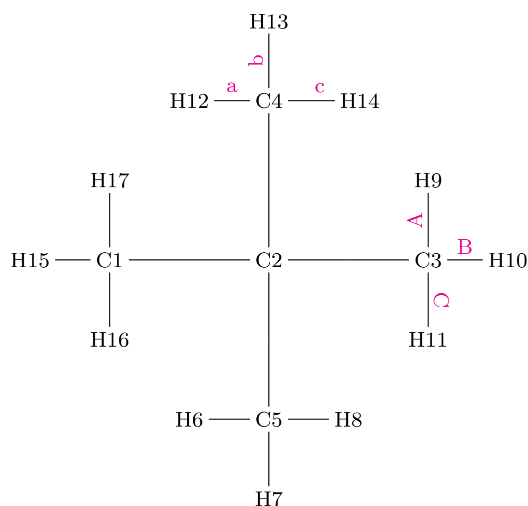


Figure 3.3: Example Graph

All twelve H atoms (labeled 6 through 17) are symmetric as are the four C atoms with labels 1, 3, 4 and 5. From this, we can infer that all twelve C-H bonds are symmetric. When $B = 1$, cutting any one of the twelve C-H bonds results in the same chemical graph with two components: (i) C_5H_{11} and (ii) H. However, when $B = 2$, we find that cutting the bond set (a, b) results in a different chemical graph from the one obtained by cutting the bond set (a, A)(Figure 3.4). Even though b and A are symmetric on the original chemical graph,

they are no longer symmetric *after* a is cut. However, bond set (a, b) is symmetric to bond set (A, B) and bond set (a, A) is symmetric to bond set (b, B). It is too expensive to rerun Bliss each time a new bond is cut. Instead, we make inferences about types of symmetry after a single run of Bliss by a more careful analysis of the Bliss output as described below.

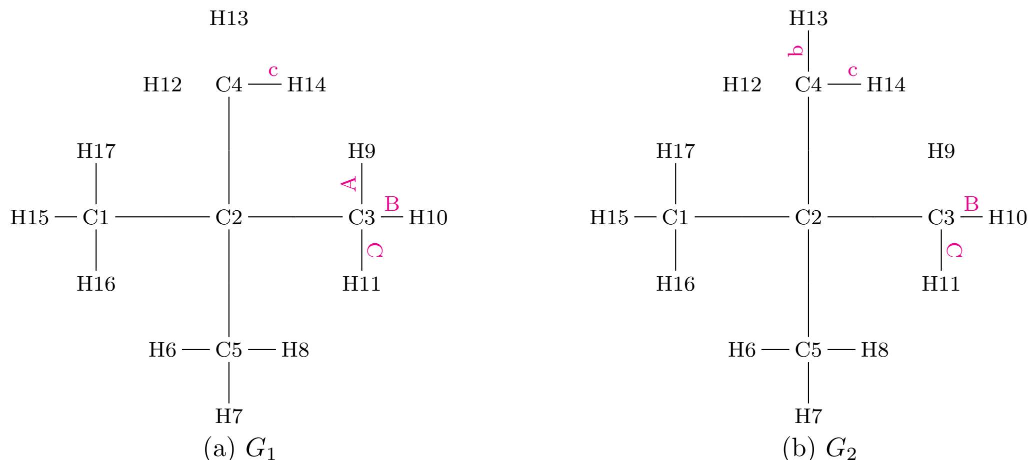


Figure 3.4: Two examples

Table 3.3 shows the output generated by Bliss for graph shown in Figure 3.3. Notice that some generators consist of a single pair of vertices such as (16, 17) whereas others such as (4, 5)(6, 12)(7, 13)(8, 14) consist of several pairs. We call the former *simple* and the latter *compound* generators. In a compound generator, the automorphism requires interchanging *all* pairs of vertices.

Consider two bond sets $BS_1 = (b_1^1, b_2^1, \cdots, b_j^1)$ and $BS_2 = (b_1^2, b_2^2, \cdots, b_j^2)$ each consisting of j bonds. Let bond b_x^y be denoted by vertex pair (u_x^y, v_x^y). *Observe that, BS_1 and BS_2 are symmetric if and only if there is a sequence (composition) of Bliss generators that maps the vertices of BS_1 to the vertices of BS_2.* More precisely, there exists a sequence of k generators g_i such that $BS_2 = g_1 \circ g_2 \circ g_3 \circ \cdots \circ g_k(BS_1)$;

i.e., $u_i^2 = g_1 \circ g_2 \circ g_3 \circ \cdots \circ g_k(u_i^1)$ and $v_i^2 = g_1 \circ g_2 \circ g_3 \circ \cdots \circ g_k(v_i^1)\ \forall\ i$;

We consider some examples below.

Table 3.3: Output from Bliss to graph shown in Figure 3.3 (labels added by us to help identify generators in the examples that follow)

Generator: (16,17)	(label: SG_1)
Generator: (15,16)	(label: SG_2)
Generator: (10, 11)	(label: SG_3)
Generator: (9,10)	(label: SG_4)
Generator: (13,14)	(label: SG_5)
Generator: (12,13)	(label: SG_6)
Generator: (7,8)	(label: SG_7)
Generator: (6,7)	(label: SG_8)
Generator: (4,5)(6,12)(7,13)(8,14)	(label: CG_1)
Generator: (3,4)(9,12)(10,13)(11,14)	(label: CG_2)
Generator: (1,3)(9,15)(10,16)(11,17)	(label: CG_3)
Nodes: 48	
Leaf nodes: 5	
Bad nodes: 0	
Canrep updates: 1	
Generators: 11	
Max level: 11	
|Aut|: $-8.7399e+245$	
Total time: 10.00 seconds	

1. Consider $BS_1 = ((1,2),(1,15))$ and $BS_2 = ((3,2),(3,10))$.

   Observe that $CG_3(BS_1)) = CG_3((1,2),(1,15)) = ((3,2),(3,9))$ and $SG_4((3,2),(3,9))$ $= ((3,2),(3,10)) = BS_2$. Thus BS_1 is symmetric to BS_2.

2. Consider $BS_1 = ((4,12),(4,13))$ and $BS_2 = ((4,12),(3,9))$.

   There is no composition of generators that map $(4,12)$ to itself while mapping $(4,13)$ to $(3,9)$. Hence BS_1 is not symmetric to BS_2.

3. Consider $BS_1 = ((2,4),(5,7))$ and $BS_2 = ((2,3),(2,5))$.

   Once again, there is no composition of generators that maps BS_1 to BS_2. Hence BS_1 is not symmetric to BS_2.

4. Consider $BS_1 = ((1,15),(1,2),(3,10))$ and $BS_2 = ((5,7),(5,2),(3,10))$.

   $BS_1 = ((1,15),(1,2),(3,10))$

$$\xrightarrow{CG_3} ((3,9),(3,2),(1,16))$$

$$\xrightarrow{CG_2} ((4,12),(4,2),(1,16))$$

$$\xrightarrow{CG_1} ((5,6),(5,2),(1,16))$$

$$\xrightarrow{SG_8} ((5,7),(5,2),(1,16))$$

$$\xrightarrow{CG_3} ((5,7),(5,2),(3,10))$$

$$= BS_2$$

Thus BS_1 is symmetric to BS_2.

The problem of finding a composition of Bliss generators (automorphisms) that map BS_1 to BS_2 can be reduced to the problem of finding a path between two vertices in a directed graph (DG) as follows: Assuming, as before, that BS_1 and BS_2 each contain j bonds. Each combination of j bonds can be denoted by a vertex in DG. Each out-edge from a vertex DG corresponds to a generator and leads to a vertex in DG corresponding to the mapping implied by the generator. Then, we a seek a directed path from the vertex corresponding to BS_1 to the vertex corresponding to BS_2. The complexity of this approach is $O(n^j g)$ where g is the number of Bliss generators and n is the number of bonds in the isomer.

In general, this is too computationally expensive for our purposes. The objective of this paper is to engineer an algorithm that quickly finds some symmetry to reduce space and time required by the SAA algorithm. From an algorithmic standpoint, the gains associated with identifying and exploiting symmetry do not justify the time required to compute all instances of symmetry. Instead, we have developed an approach for quickly finding symmetry for three commonly occurring special cases.

- Case 1: Only simple symmetry exists in graphs; (only simple generators are returned by Bliss)

  Not all compounds contain the kind of symmetry described in the example in Figure 3.3. The compounds in our current data set (Tyrosine, Nicotine, Phenmetrazine) contain only symmetry identified by simple generators. Implementation of this is discussed in detail in the next section.

- Case 2: $B = 1$

  We use transitive closure of symmetric vertices in Table 3.3 to identify vertices that are symmetric. We identify three sets of vertices, (2), (1, 3, 4, 5) and (6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16). Now, for any values of u_1, v_1, u_2, v_2 belonging to bonds b_1 and b_2, we determine whether the vertices and hence the bonds are symmetric. For example, bonds $((4,13),(3,10))$ are symmetric, while bonds $((5,8),(2,4))$ are not.

- Case 3: Special case of $B = 2$

  Let $BS_1 = (b_1^1, b_2^1)$, $BS_2 = (b_1^2, b_2^2)$ and $b_1^1 = b_1^2$. BS_1 and BS_2 are symmetric if there is a sequence (composition) of Bliss generators *not* involving u_1 or v_1 that maps the vertices of b_2^1 to b_2^2. For example, consider $BS_1 = ((4,13),(3,10))$ and $BS_2 = ((4,13),(1,15))$. We observe that $CG_3(b_2^1) = CG_3(3,10) = (1,16)$ and $SG_2(1,16) = (1,15) = b_2^1$. Thus BS_1 is symmetric to BS_2.

Figure 3.5 and Figure 3.6 illustrate the benefits of exploiting symmetry for $B = 1$ and $B = 2$, respectively for a Nicotine isomer.

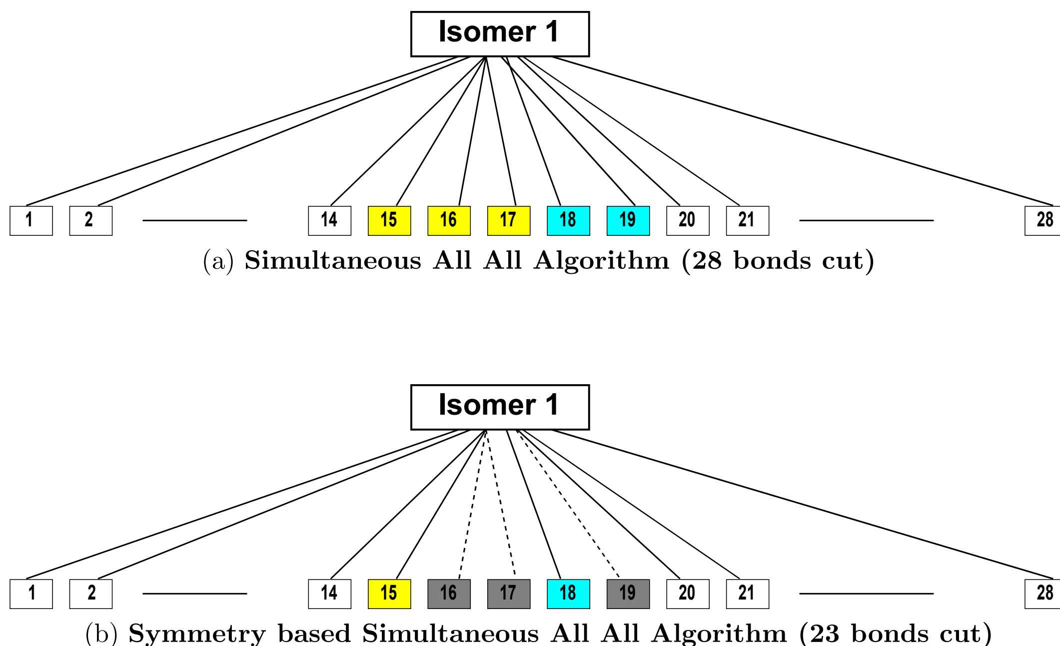
3.3 Implementation Details (Special Case 1)

Figure 3.7 is a high-level flowchart of our improved implementation. Preprocessing Steps 1 through 4 are executed for each isomer in the dataset and primarily describe the interaction with Bliss that is required to identify symmetric bonds. These are only part of the enhanced SAA algorithm (this paper). Step 5 describes the improved SAA algorithm which combines the existing SAA[1] with our enhancements. Additional details are provided in our description of Step 5.

3.4 Preprocessing Steps

Step 1: Convert input mol files to bliss input format

The Bliss input format requires a numeric color label to be explicitly associated with each vertex. To accomplish this, we parse the input mol file representing the isomer, identify the



(a) **Simultaneous All All Algorithm (28 bonds cut)**

(b) **Symmetry based Simultaneous All All Algorithm (23 bonds cut)**

Figure 3.5: Cutting one bond per isomer. In (a) we cut 28 bonds per isomer (including the symmetric ones, shown in the same color), and in (b), by incorporating the concept of not cutting more than one of a set of symmetric bonds, we cut only 23 bonds per isomer. Bonds in gray are not cut.

different types of atoms present, and assign all atoms of a specific type the same numeric color. For example, in Nicotine ($C_{10}H_{15}N_2$), all C atoms are assigned the color 0, all H atoms the color 1, and all N atoms the color 2. Figure 3.8 shows a Nicotine isomer that we will use as a running example to illustrate the individual steps of the algorithm.

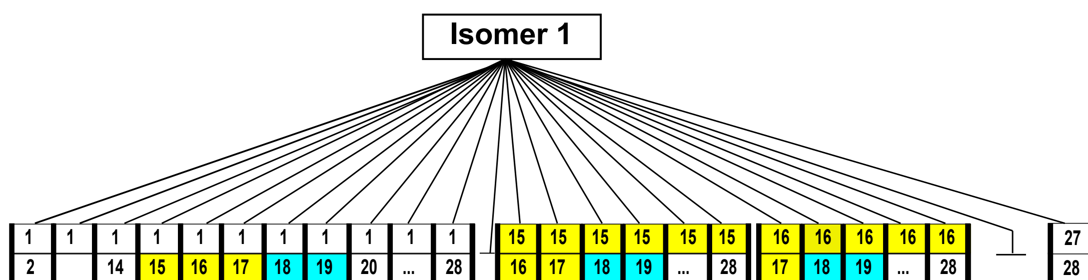
Step 2: Run files through Bliss

Running the input files through `Bliss` returns the generators for the chemical graph that will be used in Step 3 to identify symmetric vertices. For our example, `Bliss` returns the following two-vertex generators: (19, 20), (21, 22), (17,18), (15,16) and (14, 15).

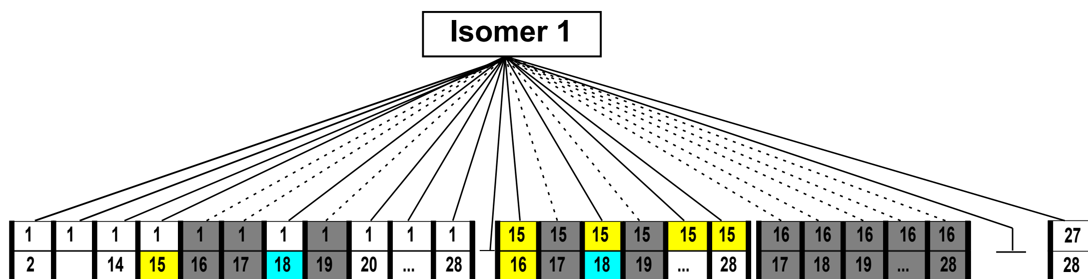
Step 3: Identify symmetric bonds in isomers using the generators returned by bliss and incorporate symmetry in input file

Figure 3.9 shows the flow chart describing Step 3.

Step 3a: Use the generators output by `Bliss` to compute all symmetric vertices. For our example, these are (19, 20), (21, 22), (17, 18), (14, 15, 16), also shown in Figure 3.10. Note



(a) **Simultaneous All All Algorithm (378 bonds cut)**



(b) **Symmetry based Simultaneous All All Algorithm (257 bonds cut)**

Figure 3.6: Cutting two bonds per isomer. In (a), we cut 378 bonds per isomer (including symmetric ones, shown in the same color), while in (b), by incorporating the concept of not cutting more than one of a set of symmetric bonds, we reduce the number of bonds cut to 257.

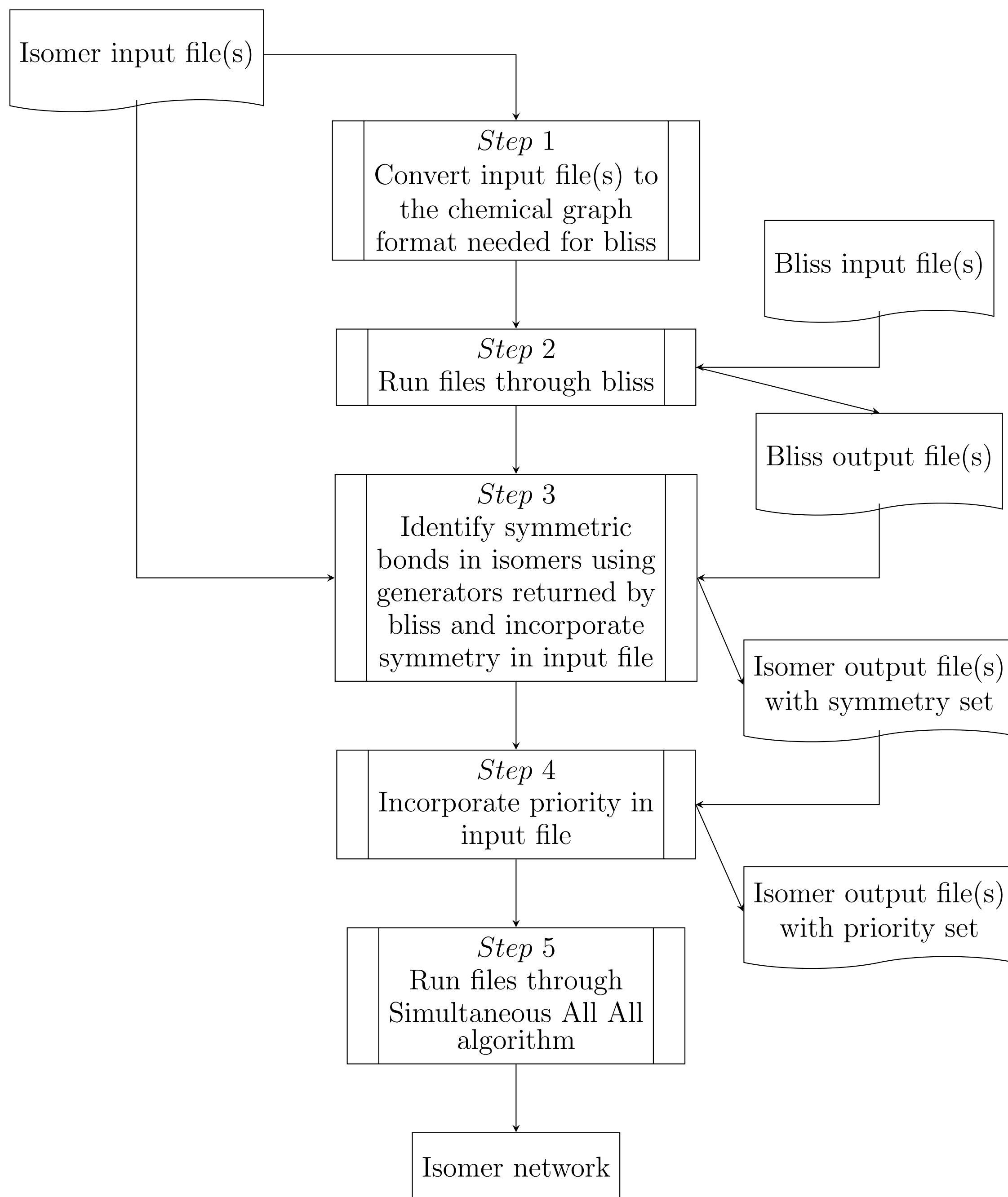


Figure 3.7: Flow chart depicting the generation of isomer networks using symmetry

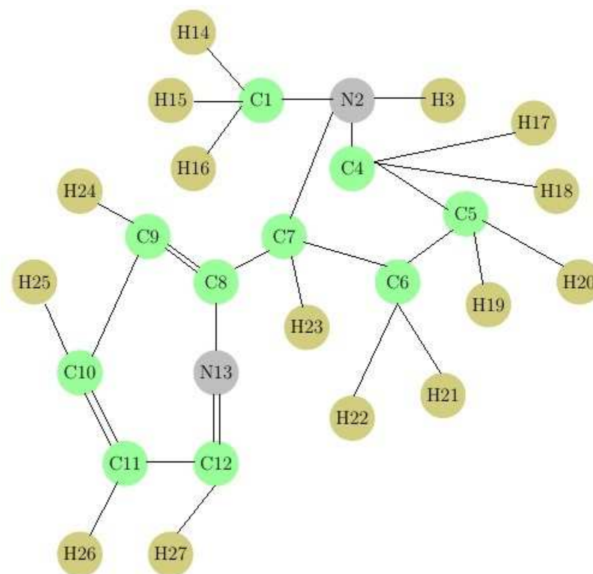


Figure 3.8: One Nicotine ($C_{10}H_{15}N_2$) isomer represented as a chemical graph

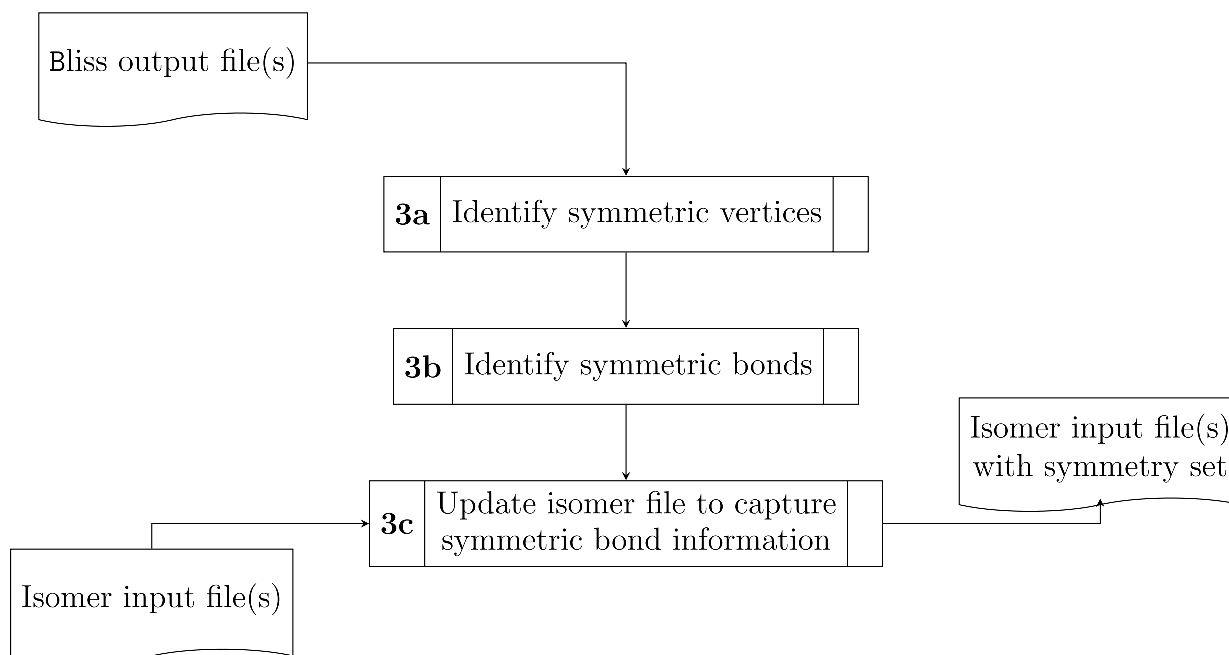


Figure 3.9: Flow chart describing Step 3

that each symmetric set is the transitive closure of the generators.

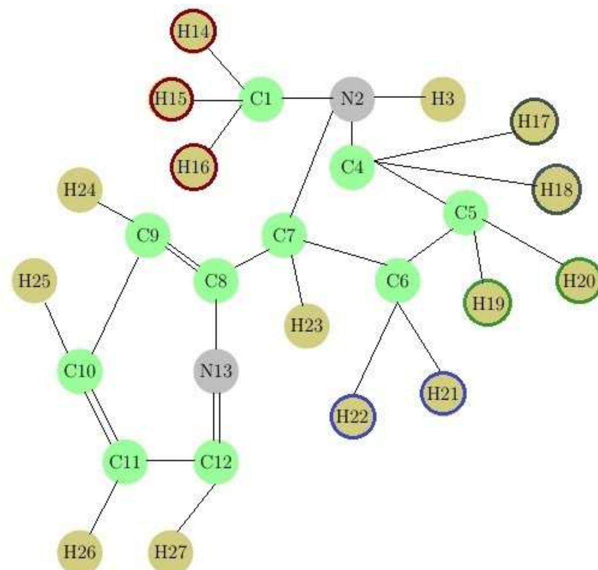


Figure 3.10: Atoms identified as symmetric following the output from Bliss

Step 3b: Use vertex symmetries to identify edge symmetries. Again, without loss of generality, two edges (u_1, v_1) and (u_2, v_2) are symmetric if $u_1 = u_2$ or u_1 and u_2 are symmetric *and* $v_1 = v_2$ or v_1 and v_2 are symmetric. Figure 3.11 shows the edge symmetries in our example.

Step 3c: We next update the input mol file from Step 1, to include information about symmetric bonds by designating one of the unused fields as the integer *symmVal* field. If several bonds are symmetric, they contain the same *symmVal*.

Step 4: Incorporate priority in input file

All symmetric bonds are assigned a distinct priority within their set for ease of processing. For example, symmetric bonds b_1, b_2 and b_3 could be assigned priority 1, 2, and 3, respectively. These are used to choose which bond to cut, in case only one or two bonds of the three need to be cut.

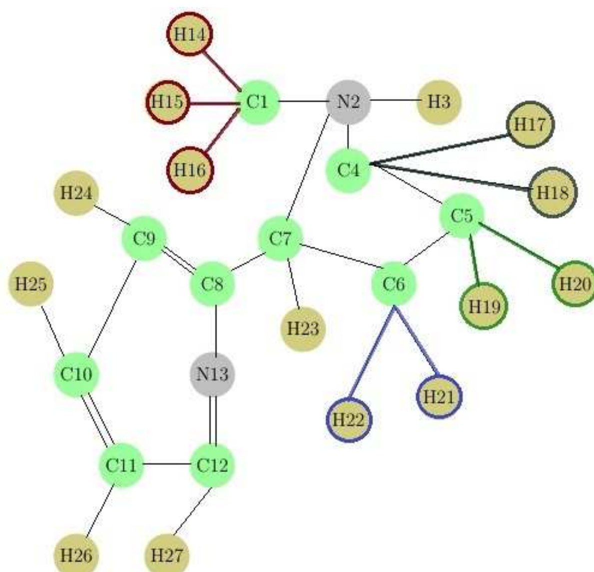


Figure 3.11: Symmetric edges identified, from symmetric vertices.

3.5 Step 5: Improved SAA

Step 5 utilizes two data structures associated with each isomer: *bitPatterns* and *dArray*. The first (*bitPatterns*) is an array that holds all possible bond subsets containing exactly j ($j = 1, 2, \cdots, B$) bonds. A bit is set to 1 if the corresponding bond is in the set and 0, otherwise. An isomer with 28 bonds for $j = 2$ results in a *bitPatterns* array with $\binom{28}{2}$ entries. Each entry is a 28 bit string with two 1s; e.g., [0000000000000000000000000011] corresponding to the set $(1, 2)$. Each bit pattern is then processed to determine *whether* the corresponding bond set is symmetric to previous bond sets (Step 5a), in which case the bond set is *not* cut. If the bond set is cut, we compute the dnName of the resulting chemical graph and store this entry in *dArray* (Step 5b). After all isomers are processed, we merge and sort the individual *dArrays*, and search for duplicates in order to determine the BCD between isomers (Step 5c).

These steps are explained further in the flow chart in Figure 3.12. New steps/ enhancements are shaded in gray.

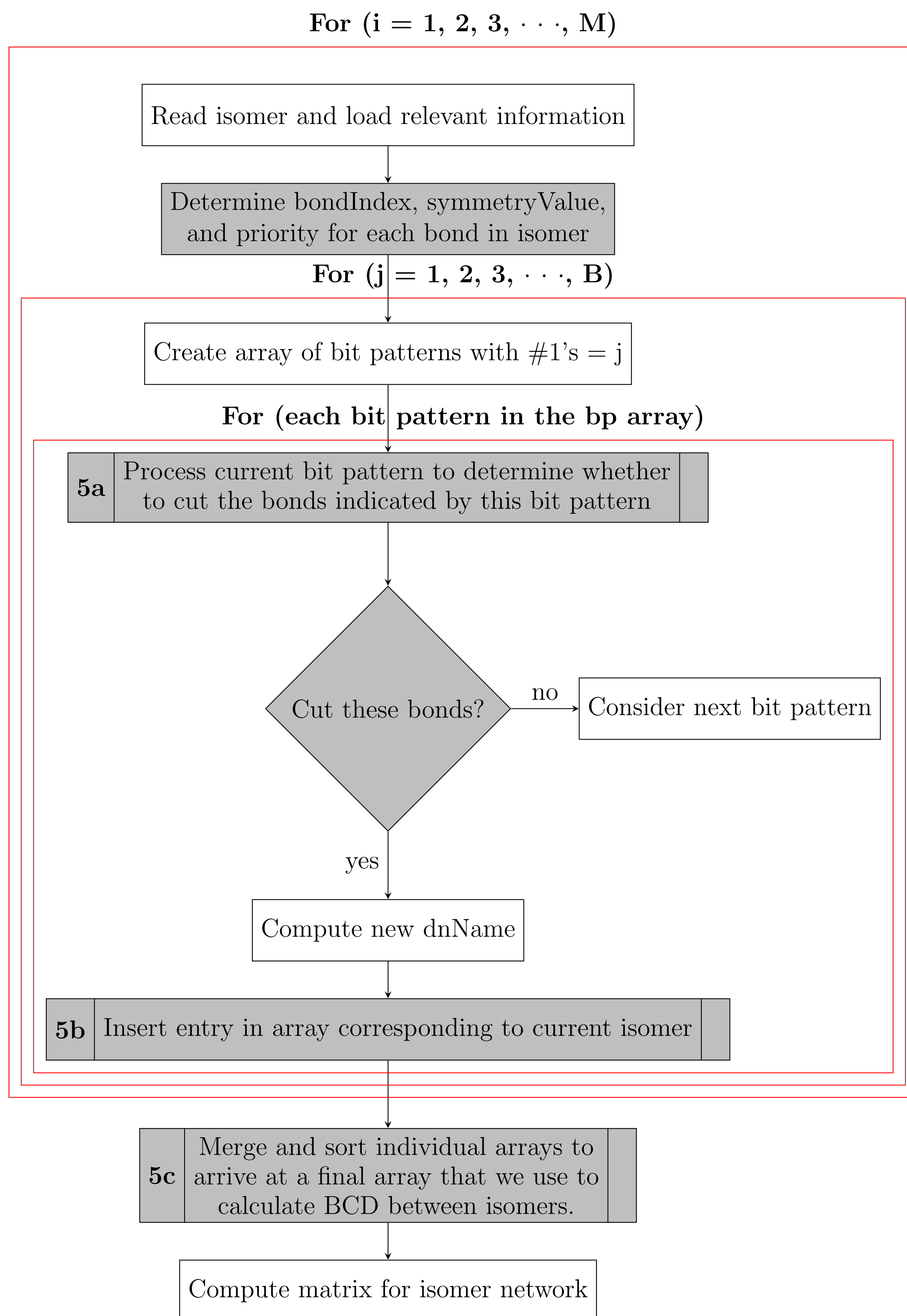


Figure 3.12: Flow chart describing Step 5 - new/modified steps in gray

Step 5a: Process current bit pattern to determine whether to cut the bonds indicated by this bit pattern

Consider the example of an isomer with 28 bonds in Table 3.4. Symmetric bonds are indicated with a $symmetryValue > 0$ and $priorityValue > 0$. Symmetry values and priority values for all other bonds are 0. Recall that we need not cut symmetric bond sets, since they generate the same dnNames, and are redundant.

Table 3.4: Sample data indicating symmetry and priority values set for bonds in the isomer file

Bond Index	Symmetry Value	Priority Value
1	1	1
2	1	2
3	2	1
4	2	2
5	3	1
6	3	2
7	3	3
8	0	0
9	0	0
···		
28	0	0

Let j be the iterator over the total number of bonds being cut (B). When $j = 1$, we only have to cut one of the bonds of in a symmetric set. We cut those with priority = 1. So we would cut bonds corresponding to bond indices 1, 3, 5, 8, 9, ···, 28. When $j = 2$, note for example that bond set $(1, 8)$ is symmetric to $(2, 8)$ or that $(1, 3)$ is symmetric to $(1, 4)$. We only need to cut one of the following: $(1, 3)$, $(1, 4)$, $(2, 3)$, $(2, 4)$. We cut bonds with priority 1, and hence will cut bond set $(1, 3)$. This approach can be used for higher values of j. In contrast, the existing algorithm [1] generates dnNames for all bond sets and discards duplicates.

Step 5b: Insert entry in $dArray$: Add the newly generated dnName for the current isomer into the $dArray$ belonging to this isomer. The main advantage relative to the previous

approach is that there is no search and insert, but only a simple $O(1)$ time append.

Step 5c: Merge and sort individual arrays to arrive at a final array that we use to calculate BCD between isomers.

3.6 Analysis

We now analyze the benefits of exploiting symmetry to reduce memory utilization for special case 1 discussed above. Let n be the total number of bonds in an isomer, B the maximum number of bonds we want to break, x the total number of symmetric bonds and y the number of instances of symmetric bonds.

When we cut up to one bond per isomer, in the existing SAA implementation we have $\binom{n}{1}$ options (i.e., generate $\binom{n}{1}$ dnNames) whereas while using symmetry, we have only $\binom{(n-x)}{1}+y$ options.

When we want to break up to two bonds, in the existing SAA implementation we have $\binom{n}{1}+\binom{n}{2}$ choices. When using symmetry we have only $\binom{(n-x)}{1}+y+\binom{(n-x+y)}{2}+y$ choices.

For example, consider a Nicotine isomer $(C_{10}H_{15}N_2)$ with 28 bonds $(B_1 \cdots B_{27})$. In this isomer, $B_1 \equiv B_2 \equiv B_3$, $B_4 \equiv B_5$, $B_6 \equiv B_7$, and $B_8 \equiv B_9$. Here, $n = 28$, $x = 9$, and $y = 4$. When we want to break upto 2 bonds per isomer, in the existing SAA implementation we break $\binom{28}{1}+\binom{28}{2} = 406$ bond sets. When using symmetry we break only $\binom{(28-9)}{1}+4+\binom{(28-9+4)}{2}+4 = 280$ bond sets per isomer.

This pattern continues up to B bonds. The exact number of options to break up to B bonds when using our streamlined approach is isomer dependent since x and y are isomer dependent. The number of options also depends on whether the isomer contains only simple or simple and complex symmetry. Note that Nicotine in our example above contains only simple symmetry.

Now consider streamlining the SAA by using an array for each isomer instead of a single $sortedArray$.

Let ${n_i}^s$ and ${n_i}^{ns}$ denote the number of dnNames generated for Isomer i, $1 \leq i \leq M$ with and without symmetric dnNames respectively. Clearly ${n_i}^{ns} \leq {n_i}^s$.

Let $N_{ns} = \sum_{i=1}^{M} n_i^{ns}$ and $N_s = \sum_{i=1}^{M} n_i^s$. Typically, $N_{ns} \ll N_s$. In the SAA implementation, a binary search is performed to check for duplicates each time a new dnName is generated ($O(\log N_s)$). If the new dnName is not a duplicate it is inserted into the sorted array ($O(N_s)$). The total complexity for inserting all N_s dnNames takes $O({N_s}^2)$ time.

In the improved implementation, a seperate array is maintained for each isomer. The cost of appending a dnName to the array for isomer i takes $O(1)$ time, since we store the position of the last element. Next, the individual arrays are merged into a single array and sorted using mergesort($O(N_{ns}\log(N_{ns}))$). The total complexity is $O(N_{ns}\log N_{ns})$, a significant improvement over $O(N_s^2)$. Note that N_s and N_{ns} are exponential in the number of bonds cut per isomer and can be quite large.

Our theoretical analysis is confirmed by the empirical results below.

3.7 Results

This section shows the results of running the algorithms before and after the enhancements presented in this paper. The SAA algorithm and enhancements have been developed in Java. All experiments were carried out on a computer running Microsoft Windows Professional 7 with a 2.67 GHz Intel Core i7 CPU and 4GB of RAM. Note that for all of the databases, we used the `Nauty` chemical graph canonical naming algorithm to test for isomorphism. We tested the code on three sets of isomers from the GDB database[5], including Nicotine isomers, Phenmetrazine isomers, and Tyrosine isomers provided to us by Professor Reymond's group.

Incorporating symmetry leads to reduction in number of bonds cut (and hence memory utilized). Memory utilized is the amount of space required to store the dnNames generated by cutting up to B bonds (B = 1, 2, 3 or 4 in our experiments) for the entire set of isomers under consideration. Recall that for each bond set cut we generate a dnName. For example, when cutting up to 4 bonds of Nicotine, the original SAA [1] breaks 603925 bonds (which results in 603925 dnNames being generated and temporarily stored). Incorporating symmetry reduces

this to 259290, thus reducing the number of dnNames generated (and thus memory utilized for dnNames).

Significant improvements in processing times were also observed as a result of our enhancements.

For example, while cutting up to 4 bonds in 25 isomers of Nicotine(Table 3.8), space utilized dropped from 205 MB to 76.5 MB, and processing time reduced from 1.59 hours to only 1.17 minutes. Similar improvements were found for Phenmetrazine (253 MB to 117 MB reduction in space utilized, and 4.52 hours to 2.56 minutes in processing time) and Tyrosine (82.9 MB to 39.9 MB in space reduction, and 30.5 minutes to 1.11 minutes in processing time). Results for these isomers while cutting up to 1, 2, 3 and 4 bonds are listed in Table 3.5, Table 3.6, Table 3.7 and Table 3.8 below. The data listed is the average of three run times.

Table 3.9 shows the speed up factors in terms of time and space for Nicotine, Phenmetrazine and Tyrosine isomers when we cut up to 1, 2, 3 and 4 bonds per isomer. We note that the greater the number of bonds we cut, the greater the improvement.

Table 3.5: Results summary when cutting **1** bond in **500** isomers

Isomer	**Nicotine**		**Phenmetrazine**		**Tyrosine**	
	Kouri et al.	This paper	Kouri et al.	This paper	Kouri et al.	This paper
Time taken	16.20 sec	7.45 sec	17.84 sec	7.67 sec	12.20 sec	7.84 sec
# Bond sets cut	14000	10900	15041	12109	12000	10145
Space utilized	4.4 MB	3.4 MB	4.98 MB	4.10 MB	3.23 MB	2.73 MB

With our enhancements we were able to break up to 2 bonds for 1500 isomers each of Nicotine, Phenmetrazine and Tyrosine, which is a significant improvement over what was possible in the existing SAA[1] using only the available RAM.

Table 3.6: Results summary when cutting up to **2** bonds in **500** isomers

Isomer	**Nicotine**		**Phenmetrazine**		**Tyrosine**	
	Kouri et al.	This paper	Kouri et al.	This paper	Kouri et al.	This paper
Time taken	21.30 min	55.83 sec	30.62 min	1.06 min	17.99 min	51.94 sec
# Bond sets cut	203000	126483	233771	155306	150000	109634
Space utilized	61.6 MB	38.5 MB	76.3 MB	50.8 MB	39.8 MB	29.1 MB

Table 3.7: Results summary when cutting up to **3** bonds in **75** isomers

Isomer	**Nicotine**		**Phenmetrazine**		**Tyrosine**	
	Kouri et al.	This paper	Kouri et al.	This paper	Kouri et al.	This paper
Time taken	27.43 min	1.03 min	44.21 min	1.21 min	15.46 min	45.81 sec
# Bond sets cut	276150	148036	339375	184600	174300	110103
Space utilized	82.5 MB	44.3 MB	109 MB	59.5 MB	45.8 MB	28.7 MB

Table 3.8: Results summary when cutting up to **4** bonds in **25** isomers

Isomer	**Nicotine**		**Phenmetrazine**		**Tyrosine**	
	Kouri et al.	This paper	Kouri et al.	This paper	Kouri et al.	This paper
Time taken	1.59 hours	1.17 min	4.52 hours	2.56 min	30.50 min	1.11 min
# Bond sets cut	603925	259290	798250	368648	323750	155440
Space utilized	205 MB	76.5 MB	253 MB	117 MB	82.9 MB	39.9 MB

Table 3.9: Improvement factor in terms of time and space

Isomer	**Nicotine**		**Phenmetrazine**		**Tyrosine**	
	Time	Space	Time	Space	Time	Space
up to 1 bond set cut	2.17	1.30	2.33	1.21	1.56	1.18
up to 2 bond sets cut	22.90	1.60	28.89	1.50	20.78	1.37
up to 3 bond sets cut	26.63	1.86	36.54	1.83	20.24	1.60
up to 4 bond sets cut	81.54	2.68	105.94	2.16	27.48	2.07

CHAPTER 4

GROUPING ISOMERS

As explained in previous chapters, the extraction of isomer networks is a time and data-intensive computation. Despite streamlining the SAA algorithm and utilizing symmetry, it is not possible to reduce the amount of intermediate memory needed to the extent that we are able to process all of the dnNames of all of the isomers at one time. For example, we estimate that computing the bond count distance (BCD) of all the 1,050,219 isomers of Nicotine will require 5 TB. In this chapter, we approach this problem by considering isomers in smaller groups and applying the All-Pairs SAA algorithm to each group. This will compute the BCD between all pairs of isomers in that group. When all groups are so processed, we will have computed the BCD between all isomers. However, creating these groups efficiently turns out to be non-trivial and is the main contribution of this chapter.

The problem can also be described colloquially as follows: we wish to have a party for M friends so that everyone gets to shake hands with everyone else. However, our house can only hold $N << M$ people, so we propose to solve this problem by holding several smaller parties (each of size N) so that any pair of friends meets in exactly one party. What is the minimum number of parties we need to have and whom should we invite to each party? Suppose $M = 9$ with individuals $\{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8\}$ and $N = 3$. A possible solution to this is given by $\{I_0, I_1, I_2\}$, $\{I_3, I_4, I_5\}$, $\{I_6, I_7, I_8\}$, $\{I_0, I_3, I_6\}$, $\{I_1, I_4, I_7\}$, $\{I_2, I_5, I_8\}$, $\{I_0, I_5, I_7\}$, $\{I_1, I_3, I_8\}$, $\{I_2, I_4, I_6\}$, $\{I_0, I_4, I_8\}$, $\{I_1, I_5, I_6\}$ and $\{I_2, I_3, I_7\}$.

The isomer-network construction example (Figure 4.1 shows nine isomers I_0 through I_8) whose Nauty name (NN) lists must be present in memory at a single time. If instead only three isomers' NN names can be accommodated at a time, we consider and run the All-Pairs SAA algorithm on three isomers. This will compute the BCD between all pairs of isomers in that group. Figure 4.2 shows a snapshot of this approach for isomers I_0 though I_8. We run

12 iterations of the All-Pairs SAA (four of which are shown) with groups of three isomers in each iteration. These 12 iterations can be run sequentially or in parallel. When all groups are processed, we will have computed the BCD between all isomers.

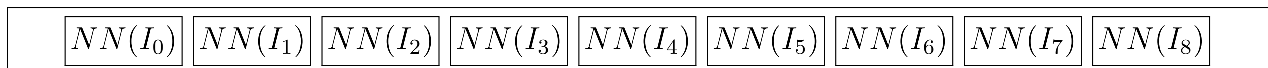
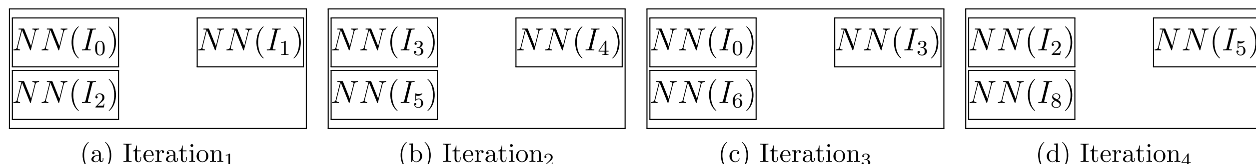


Figure 4.1: All Nauty names for nine isomers in memory



(a) Iteration_1 (b) Iteration_2 (c) Iteration_3 (d) Iteration_4

Figure 4.2: Four of 12 iterations using the grouping technique

We now present a formal description of the problem.

Problem Statement: Let M denote the number of isomers on which we wish to compute the isomer network and N the number of isomers that can be processed in memory at a given time. Let $\{I_0, I_1 \ldots I_{M-1}\}$ denote the M isomers. Our goal is to generate a list of groups of isomers where each group contains N isomers ($N << M$) such that all pairs of isomers (I_x, I_y) with $1 \leq x < y \leq N$ belong to the same group once and only once. Clearly, a pair of isomers must belong to the same group at least once in order to compute the BCD between them. In addition, an efficient solution will avoid placing any pair of isomers in the same group more than once to avoid the redundancy of computing the BCD between them multiple times.

The correctness and efficiency of our algorithm requires N to be a prime number. In practice, N can be chosen to be any prime number that permits an in-memory run of the SAA algorithm. Given N, we determine the smallest d such that $M \leq N^d$. If $M < N^d$, we pad the input with $M - N^d$ dummy isomers. These are used during the generation of groups, but are discarded in the all-pairs BCD calculations. Our grouping algorithm is

recursive (inductive) with respect to d.

We begin with a description of the two base cases $d = 2$ and $d = 3$. (Note that the case where $d = 1$ is not interesting because it implies that $M = N$.) We later describe how to generalize the algorithm to higher values of d. While the algorithms described here were developed independently by us, we have subsequently seen references to the 2D version of the problem in recreational mathematics in the context of forming groups of people; e.g., in [23].

Our approach has roots in combinatorial designs and more specifically Balanced Incomplete Block Designs (BIBD). A BIBD can be represented as a design of the form (v, b, r, k, λ) where v is the total number of elements in the block domain (X), b is the number of blocks and k is the size of each block. Also, each element x_j has the same valence (i.e., each appears in the same number r of blocks) and each pair of elements x_i and x_j have the same covalence (i.e., appear in the same number λ of blocks). r is called the replication number and λ is called the index of the design[24]. Note that in a BIBD every block is the same size $k \geq 2$. A balanced design is complete if $k = v$, so that each block contains all v. If $k < v$ (as in our case), B is incomplete. [24, 25]

Consider the example, $X = \{0, 1, 2, 3\}$ with blocks $B_1 : 012, B_2 : 013, B_3 : 023, B_4 : 123$.[24] Note that, $v =\mid X \mid= 4$, $b = 4$ (total number of blocks), $r = 3$ (the number of blocks each element appears in), $k = 3$ (size of each block) and $\lambda = 2$ (number of blocks each pair of elements appear in - i.e, 01 appears in 2 blocks, 12 appears in 2 blocks and so forth).
Thus, X and the blocks B_1, B_2, B_3 and B_4 form a $(v = 4, b = 4, r = 3, k = 3, \lambda = 2)$ - design.

Our problem has N^d elements. We want each block (k) to contain N elements each. Since no two elements should be in the same group more than once, λ has to be one. Given fixed values of v, k, and λ, we arrive at $b = (\frac{N^d - 1}{N - 1} \times N^d)$ and $r = (\frac{N^d - 1}{N - 1})$. The derivation for the total number of blocks (b) and number of blocks each element will be a part of (r) is explained later in the paper.
Thus our problem forms a $(v = N^d, b = (\frac{N^d - 1}{N - 1} \times N^d), r = (\frac{N^d - 1}{N - 1}), k = N, \lambda = 1)$ - BIBD.

We are not aware of algorithms or software solutions to solve a general BIBD problem. Our algorithm solves the problem for cases where v is the power of a prime number (N), $k = N$ and $\lambda = 1$.

4.1 Two dimensional case ($d = 2$)

In this section we start with an example, followed by the proof of correctness for the case $d = 2$.

4.1.1 Example

We describe the algorithm along with a running example where $N = 3$ and $d = 2$ (i.e., $M = 9$). We visualize the nine isomers as being organized in a 3×3 matrix as shown in Figure 4.3.

R_0	0	1	2
R_1	3	4	5
R_2	6	7	8

Figure 4.3: Matrix of isomers

Figure 4.4 shows the steps in the algorithm for case $d = 2$.

Step 0: Each row R_i, $0 \leq i < N$ of isomers is output as a group.

Step 1: This step alternately iterates through Steps 1a and 1b below N times.

- **Step 1a:** Output each column of isomers as a group.
- **Step 1b:** Perform a circular shift within each row; specifically, shift Row R_i by i elements. Thus, row R_0 remains unchanged, row R_1 is circularly shifted by 1, row R_2 is circularly shifted by 2, etc.

Figure 4.4: 2D algorithm

On our example, **Step 0** outputs three groups (0, 1, 2), (3, 4, 5), (6, 7, 8). In the first of the N (three) iterations of **Step 1**, **Step 1a** outputs (0, 3, 6), (1, 4, 7), and (2, 5, 8) while **Step 1b** performs the data movement shown in Figure 4.5.

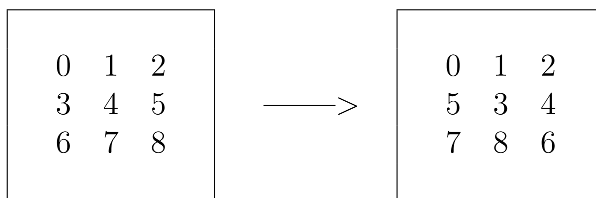


Figure 4.5: Matrices before and after the first circular shift

In the second iteration, **Step 1a** outputs (0, 5, 7), (1, 3, 8), and (2, 4, 6) while **Step 1b** performs the data movement shown in Figure 4.6.

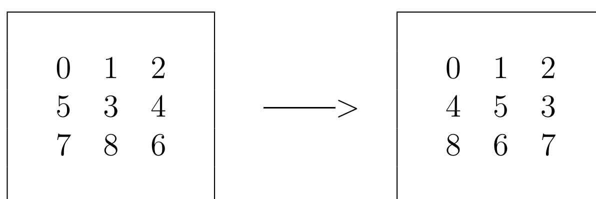


Figure 4.6: Matrices before and after the second circular shift

In the third iteration, **Step 1a** outputs the groups (0, 4, 8), (1, 5, 6), (2, 3, 7) and **Step 1b** performs a data movement that restores the original matrix shown in Figure 4.3. The algorithm is now complete and the list of groups so obtained can be used to compute the BCD between all pairs of isomers.

Table 4.1 shows the exact step in which each pair of isomers meet in a group. An S_0 entry denotes that the isomers met each other in **Step 0** and an S_i entry, for $1 \leq i \leq 3$ indicates they met in iteration i of **Step 1a**. Note that **Step 1b** simply moves data without outputting any groups and is not represented in the table.

Table 4.1: All-pairs table (matrix is symmetric, so lower triangle is not shown) showing the algorithmic step where each pair of isomers meets.

		Isomer Number								
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	0	X	S_0	S_0	S_1	S_3	S_2	S_1	S_2	S_3
	1		X	S_0	S_2	S_1	S_3	S_3	S_1	S_2
	2			X	S_3	S_2	S_1	S_2	S_3	S_1
	3				X	S_0	S_0	S_1	S_3	S_2
	4					X	S_0	S_2	S_1	S_3
Isomer	**5**						X	S_3	S_2	S_1
Number	**6**							X	S_0	S_0
	7								X	S_0
	8									X

4.1.2 Implementation Details

Our implementation utilizes a one-dimensional row-major order representation of the 2D matrix rather than a two-dimensional array. Although this complicates the implementation of the data movement operations, it gives us the flexibility to switch between different combinations of N and d depending on M and memory capabilities of our system. We now describe functions that will be used subsequently for the cases where $d > 2$.

sequentialGroups(): implements **Step 0**.

process2D(): implements all of **Step 1**.

The 2D algorithm may be viewed simply as a call to sequentialGroups() followed by a call to process2D().

4.1.3 Proof of correctness

Theorem 1. *The two-dimensional algorithm described in Figure Figure 4.4 is correct; i.e., any two isomers appear together in exactly one of the output groups.*

Proof. Consider any two isomers I_α and I_β. Let I_α be at location $(i,\, j)$ and I_β at location $(k,\, l)$ in the original $N \times N$ matrix. If $i = k$, I_α and I_β are in the same row and "meet" once (are placed in the same group) in Step 0. We next consider the case where $i \neq k$ (i.e., I_α and I_β are initially in different rows) and show that they meet exactly once in one of the iterations of Step 1. If $j = l$, I_α and I_β are initially in the same column, and meet in the first iteration of Step 1 (i.e., after 0 data movement steps).

The remainder of the proof addresses the case where $i \neq k$ **and** $j \neq l$. We first observe that the column number of isomer I_α after x data movement steps is given by the expression $(j + ix) \mod N$. Here j is its initial column number and x is multiplied by the row number i because the size of each of the x circular shifts is the row number i. Similarly, the column number of isomer I_β after x steps can be computed as $(l + kx) \mod N$. In order to determine whether and when I_α and I_β meet (i.e., they are in the same column after the same number of steps), we must solve the following equation.

$$(j + ix) \mod N \equiv (l + kx) \mod N$$

Rearranging terms, we get

$$\begin{aligned}
(i - k)x &\equiv (l - j) \mod N \\
(i - k)x &\equiv (N - (j - l)) \mod N
\end{aligned}$$

This gives us a linear congruence equation of the form

$$ax \equiv b \mod m, \tag{4.1}$$

where $a = i - k$ (without loss of generality, we assume $a > 0$), $b = (N - (j - l))$, and $m = N$. Let $d = gcd(a, m)$. Equation 4.1 is solvable iff $d|b$ (b is divisible by d) and has d solutions. Section 4.6 in[26] In this case, $d = 1$ (since N is prime and $a < N$). Hence Equation 4.1 is solvable and has exactly one solution.

□

4.2 Three-dimensional example ($d = 3$)

We extend our results from the previous section to three dimensions with an example that will help in understanding the algorithm presented in the next section. A similar approach will later be used to extend to greater than three dimensions. Consider $N = 3$ and $d = 3$ (i.e. $M = 27$). We visualize the data as three rows and three columns (Table 4.2), later grouped into three 3×3 columns (C_0, C_1, and C_2) as shown in Figure 4.7. The reason will become evident in Section 4.3 as we discuss higher d i.e. ($d > 2$). Elements in each C_i are of a different color to keep track of the original column they belonged to as we proceed further in the example.

Table 4.2: 27 elements are divided into 3 rows and 3 columns.

	C_0			C_1			C_2		
R_0	0	1	2	3	4	5	6	7	8
R_1	9	10	11	12	13	14	15	16	17
R_0	18	19	20	21	22	23	24	25	26

C_0			C_1			C_2		
0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26

Figure 4.7: Original $3 \times 3 \times 3$ matrix

The 3D algorithm is described below. Steps 0 and 1 independently execute the corresponding steps of the two-dimensional algorithm of the previous section *for each* C_i in Figure 4.7.

Step 0: For each C_i (in Figure 4.7), execute sequentialGroups(). In the example, we add (0, 1, 2), (3, 4, 5), (6, 7, 8), (9, 10, 11), (12, 13, 14), (15, 16, 17),(18, 19, 20), (21, 22, 23), (24, 25, 26) to our list of groups.

Step 1: For each C_i, execute process2D(). In the first iteration, this adds (0, 9, 18), (1, 10, 19), ..., (7, 16, 25), (8, 17, 26) to the list. The next two iterations add (0, 11, 19), (1, 9, 20), (2, 10, 18) ... (6, 17, 25), (7, 15, 26), (8, 16, 24) to the list.

Step 2: This is the key step of the 3D algorithm that extends the circular shift concept previously discussed in Section 4.1 to three dimensions: specifically *blocks* of three elements are now circularly shifted among C_i. The starting point is the original matrix in Figure 4.7.

In this step all of the elements in row i in each column are circularly right-shifted i columns. Thus, all of the elements in Row 2 in C_1 are moved to the corresponding locations in C_3. After each data movement operation, we execute process2D() on each column to ensure that elements in different rows of the column meet each other. This is performed $N-1$ times (twice in our example). The two configurations are shown in Figure 4.8 and Figure 4.9. A final data movement operation restores the original 3D matrix.

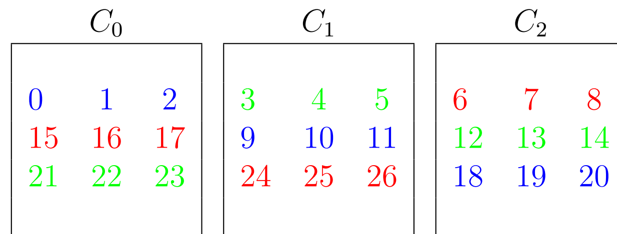


Figure 4.8: Matrix after the first data movement

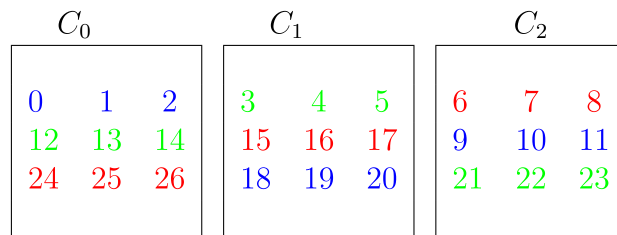


Figure 4.9: Matrix after the second data movement

Step 3: Combine all of the elements in Row i of all N columns into a single 2D column, for all i; e.g., Row 0 of all of the original C_i are combined to form a new 2D column, Row 1

of the original C_i are similarly combined, etc. This step results in N 2D columns as shown in Figure 4.10.

C_0			C_1			C_2		
0	1	2	9	10	11	18	19	20
3	4	5	12	13	14	21	22	23
6	7	8	15	16	17	24	25	26

Figure 4.10: Rows 0, 1, and 2 form individual C_0, C_1, and C_2

Then, execute process2D() on each of the three C_i in Figure 4.10. Note that we do not execute sequentialGroups() because all of the individual rows have already been output in **Step 1**. This step adds (0, 3, 6), (1, 4, 7), (0, 5, 8), (1, 3, 8), (2, 4, 6) ... (18, 23, 25), (19, 21, 26), (20, 22, 24) to our list of groups.

4.2.1 Proof of correctness

Theorem 2. *The three-dimensional algorithm described above ensures that any two isomers appear together in exactly one of the output groups.*

Proof. Two isomers in the same row of the same column (C_i) in the original configuration meet in **Step 0**. Two isomers in different rows in the same column in the original configuration meet in **Step 1** - this follows from the correctness of process2D(). Two isomers in the same row of *different* columns meet in **Step 2**. This follows from the data movement performed in **Step 2** and the correctness of process2D(). This leaves the case where two isomers are in *different* rows in *different* columns in the original configuration. A number-theoretic argument similar to that used in the proof of Theorem 1 guarantees that **Step 3** will place different rows from different columns in the same column exactly once; that isomers contained in these different rows are guaranteed to meet follows from the correctness of process2D(). □

4.3 Higher dimensional case (d>2)

Data is initially stored in a one-dimensional array as shown in Table 4.3. The total number of isomers (elements) is $M = N^d$.

Table 4.3: Data (N^d elements) stored as a one-dimensional array ($mainArray$)

Index	[0]	[1]							$[N^d - 1]$
Element	0	1	2						$N^d - 1$

In the first step (**Step 0** from Section 4.1), elements in groups of N are output. If $d = 2$, we call Process2D() (see Section4.1) and exit. When $d > 2$, we divide the elements into $N \times N$ blocks B_{ij}, $0 \leq i, j < N$, with each B_{ij} containing N^{d-2} elements in the range $[iN^{d-1} + jN^{d-2}, iN^{d-1} + (j + 1)N^{d-2} - 1]$ (See Table 4.4).

Row R_i is defined as $R_i = \bigcup_{j=0}^{N-1} B_{ij}$ and column C_j is defined as $C_j = \bigcup_{i=0}^{N-1} B_{ij}$. Clearly each row/column contains N^{d-1} elements.

Table 4.4: N^d elements of Table 4.3 are divided into N rows and N columns.

	C_0	C_1		C_j	C_{N-1}
R_0					
R_1					
R_i				B_{ij}	
R_{N-1}					

Consider an example where $N = 3$ and $d = 5$.(Table 4.5)

Dividing $243 = 3^5$ elements into 3×3 blocks results in each B_{ij} consisting of $3^{5-2} = 27$ elements and each row/column consisting of $3^{5-1} = 81$ elements as shown in Table 4.6.

Table 4.5: Data (243 elements) stored as a one-dimensional array ($mainArray$)

Index	[0]	[1]							[242]
Element	0	1	2						242

Table 4.6: 243 elements of Table 4.5 are divided into 3 rows and 3 columns.

	C_0			C_1			C_2		
R_0	0	$\cdots$	26	27	$\cdots$	53	54	$\cdots$	80
R_1	81	$\cdots$	107	108	$\cdots$	134	135	$\cdots$	161
R_2	162	$\cdots$	188	189	$\cdots$	215	216	$\cdots$	242

Next, we alternate recursive processing of each column with a circular shift of blocks. Similar to Step 1 in Figure 4.4, this is repeated N times, ensuring that blocks from different rows are placed in the same column exactly once. The block circular shifts are depicted in Table 4.7. The Nth shift will bring us back to the original matrix.

We now describe the recursive processing of each column by continuing our example from Table 4.6. The elements of C_0 in Table 4.6 are reorganized as Table 4.8. (Recall that the matrix representation is only conceptual. Data is actually stored as a one-dimensional array.)

C_0 has $3^4 = 81$ elements. C_0 is divided into $N \times N$ blocks, so that each column contains $N^3 = 27$ elements. This is shown in Table 4.9.(Table 4.9 shows only division of C_0. Similar processing is carried out for C_1 and C_2).

We repeat the recursive process for each C_j until we have N^2 (9) elements in each column (Table 4.10) when we call Process2D() on each C_j.

Once elements in groups of N^2 are processed ($C_{000}, C_{001}, C_{002}$ in Table 4.10), we perform a circular shift (Table 4.11) of elements and process every column after each shift.

Circular shift 3 will bring elements back to their original places (Table 4.10). After circular shift 3, all elements in Column C_{00} of Table 4.9 have been in a group together once and only once. We repeat the same process for Columns C_{01} and C_{02} of Table 4.9. Circular

Table 4.7: Concept of circular shift

Initial data for circular shift

	C_0	C_1	C_2	C_3			C_{N-1}
R_0	?	*	+	-		/	//
R_1	?	*	+	-		/	//
R_2	?	*	+	-		/	//
R_{N-1}	?	*	+	-		/	//

Circular shift 1

	C_0	C_1	C_2	C_3			C_{N-1}
R_0	?	*	+	-		/	//
R_1	//	?	*	+	-		/
R_2	/	//	?	*	+ -		
R_{N-1}	*	+	-		/	//	?

Circular shift N-1

	C_0	C_1	C_2	C_3			C_{N-1}
R_0	?	*	+	-		/	//
R_1	*	+	-		/	//	?
R_2	+	-			/ //	?	*
R_{N-1}	//	?	*	+	-		/

Table 4.8: Data stored as a one-dimensional array (C_0)

Index	[0]	··	[26]	[27]	··	[53]	[54]	··	[80]
Element	0	··	26	81	··	107	162	··	188

Table 4.9: 81 elements in Table 4.8 are further divided into 3 rows and 3 columns.

	C_{00}			C_{01}			C_{02}		
R_{00}	0	$\cdots$	8	9	$\cdots$	17	18	$\cdots$	26
R_{01}	81	$\cdots$	89	90	$\cdots$	98	99	$\cdots$	107
R_{02}	162	$\cdots$	170	171	$\cdots$	179	180	$\cdots$	188

shifts (of elements in B_{0i0j} where $(0 \leq i, j \leq N-1)$) and subsequent recursive processing of columns C_{00}, C_{01}, and C_{02} in Table 4.9 ensures all elements in C_0 of Table 4.6 have been processed and grouped together. Similar recursive processing is employed on elements in columns C_1 and C_2 of Table 4.6 which is followed by circular shifts of elements in B_{ij}. After circular shift 3 of elements in Table 4.6, all elements in each of the columns have been grouped together.

Note that elements within the same row (for example, elements 0 through 80, 81 through 161) have not been processed together. To take care of this, we repeat a similar process (as described for columns above) for each of the rows R_i $(0 \leq i \leq (N-1))$.

We do not want elements to meet each other more than once. We avoid this by ensuring indexNo $\neq$ elementNo in our recursive calls. This is handled once in **Step 1** of Process2D() after which if indexNo = elementNo, we have already processed set of data under consideration.

4.4 Implementation

Figure 4.11 describes the steps that help group N^d elements into groups of N elements each.

Table 4.10: 27 elements of column C_{00}in Table 4.9 are further divided into 3 rows and 3 columns.

	C_{000}			C_{001}			C_{002}		
R_{000}	0	1	2	3	4	5	6	7	8
R_{001}	81	82	83	84	85	86	87	88	89
R_{002}	162	163	164	165	166	167	168	169	170

Table 4.11: Example of circular shift

Circular shift 1 of elements in Table 4.10

	C_{000}			C_{001}			C_{002}		
R_{000}	0	1	2	3	4	5	6	7	8
R_{001}	87	88	89	81	82	83	84	85	86
R_{002}	165	166	167	168	169	170	162	163	164

Circular shift 2 of elements in Table Table 4.10

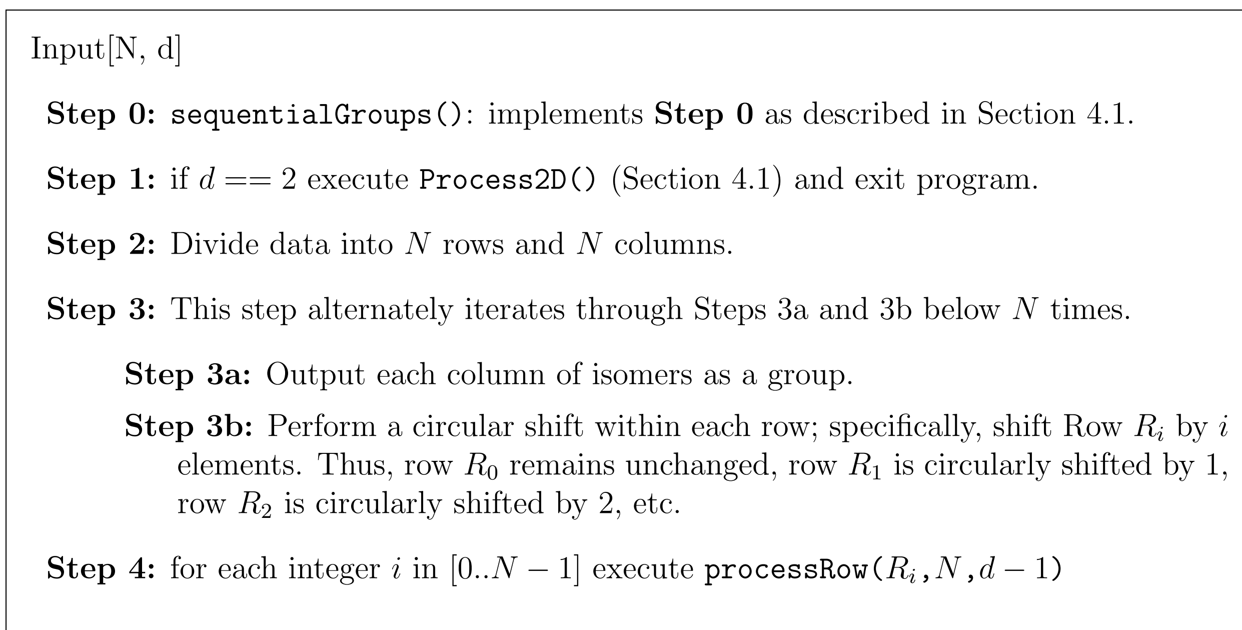
	C_{000}			C_{001}			C_{002}		
R_{000}	0	1	2	3	4	5	6	7	8
R_{001}	84	85	86	87	88	89	81	82	83
R_{002}	168	169	170	162	163	164	165	166	167

Processing columns (described in the previous section, implemented in processColumn()) and processing rows (implemented in processRow()) only differ in how data in the $mainArray$ is split into C_js.

In the case of processColumn(), each C_j contains elements as shown in Table 4.4. Recall that $C_j = \bigcup_{i=0}^{N-1} B_{ij}$. Each B_{ij} contains N^{d-2} elements starting at $(i * N^{d-1})$.

In case of processRow() we sequentially divide data into N^{d-1} elements. The algorithm for processColumn() is shown in Figure Figure 4.12. The algorithm for processRow() is not provided, as it follows similar steps as processColumn(), except for how data is divided.

We use certain auxiliary dynamic structures to enable us to shift integers (during circular shift). We do not go into further detail here.



Input[N, d]

Step 0: `sequentialGroups()`: implements **Step 0** as described in Section 4.1.

Step 1: if $d == 2$ execute `Process2D()` (Section 4.1) and exit program.

Step 2: Divide data into N rows and N columns.

Step 3: This step alternately iterates through Steps 3a and 3b below N times.

Step 3a: Output each column of isomers as a group.

Step 3b: Perform a circular shift within each row; specifically, shift Row R_i by i elements. Thus, row R_0 remains unchanged, row R_1 is circularly shifted by 1, row R_2 is circularly shifted by 2, etc.

Step 4: for each integer i in $[0..N-1]$ execute `processRow(`R_i`,`N`,`$d-1$`)`

Figure 4.11: Steps to list isomer numbers in groups of N

4.5 Deriving number of groups of isomers

There are $M = N^d$ elements. Each element has to be in the same group as the other $N^d - 1$ elements once and exactly once. Hence each element has to be in $\frac{N^d-1}{N-1}$ groups. Since there are N^d elements, there will be $\frac{N^d-1}{N-1} \times N^d$ groups. Since each group has N elements, the total number of groups can be given by $\frac{\frac{N^d-1}{N-1} \times N^d}{N}$. Simplifying this we get, $\frac{(N^d-1) \times N^{d-1}}{N-1}$ groups. When $N = 3$, and $d = 2$ we get a list of $\frac{8 \times 3}{2} = 12$ groups.

4.6 Application

Figure 4.13 is a representation of our grouping algorithm.

The list of groups (G) output by the algorithm contains $\frac{(N^d-1) \times N^{d-1}}{N-1}$ groups each of size N.

This list is used by the SAA algorithm to process isomers and generate the isomer network. Figure 4.14 lists the steps the SAA follows for any group of N isomers, where processing

1: **procedure** PROCESSCOLUMN(Array C_i, N, d)
2: Divide data into N columns (Table 4.4)
3: (C_0, C_1, $\cdots$, C_{N-1})
4:
5: **for** each integer i in $[0..N-1]$ **do**
6: **if** $d == 2$ **then**
7: **if** (!in same block) **then**
8: `Process2D(`$subArray_i$`)`
9: **end if**
10: **else**
11: processColumns($inpArr_i$, N, $d-1$)
12: perform circular shift of blocks in $subArray_i$
13: **end if**
14: **end for**
15: **for** each integer $mainCnter$ in $[0..N-2]$ **do**
16: ▷ We only need N-2 more block shifts for all
17: ▷ elements to be in the same block as each other
18: perform circular shift of blocks in $mainArray$
19: processColumns($mainArray$, N, d)
20: **end for**
21: **end procedure**

Figure 4.12: Process processColumns()

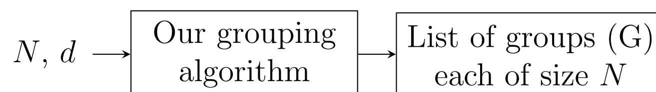


Figure 4.13: Grouping isomers

of all N isomers fits into memory.

Step a: Read N isomer input files (.mol) files

Step b: Generate a canonical name[12, 13] after *every* bond is broken (up to $B = 3$ bonds) for *each* isomer. This is the most complicated step in the SAA (in terms of set up required, time taken, and memory resources used). For any combination of isomer I_x and bonds cut B, the output at this step will always be the same.

Step c: Merge and sort all canonical names for all isomers.

Step d: Compute BCD by checking for duplicates in our merged and sorted list of canonical names. If two isomers I_x and I_y have the same canonical names (obtained after breaking b_x and b_y number of bonds for I_x and I_y respectively), then $BCD(I_x, I_y) = b_x + b_y$.

Step e: Update isomer network. The isomer network is stored as a file with entries in the form $I_x \quad I_y \quad BCD(I_x, I_y)$ where x and y belong to the group of isomers being processed.

Figure 4.14: List of steps executed by SAA algorithm

After the execution of the All Pair SAA is complete, our isomer network file contains the BCD between each and every pair of isomers processed by SAA such that $2 \leq BCD \leq 2B$, where B is the maximum number of bonds per isomer we break. We have a lower limit of 2, since we break atleast one bond per isomer.

Recall, we want to find all pair Bond Count Distances between M isomers. Only N isomers can be processed in memory at one time ($N^d \leq M$). If $N^d < M$, and our groups have dummy isomer entries, these dummy values are ignored as part of the input to the SAA algorithm. Each of the groups (set of groups) can be processed in parallel to finally arrive at a file that contains all pairs Bond Count Distances of all M isomers.

We now present two approaches to generating a network of M isomers.

4.6.1 In-memory approach

In this approach we simply run the improved All Pairs SAA algorithm with the N isomers in each group $\dfrac{(N^d - 1) \times N^{d-1}}{N - 1}$ times.

The algorithm iterates through the entire list of isomer groups and stores/appends the output of each iteration to the same output file (Figure 4.15). Each iteration of the SAA algorithm will find the all-pairs BCD between N isomers within a single group. The output file contains entries in the form $I_x \quad I_y \quad BCD(I_x, I_y)$ where x and y belong to the group of isomers being processed during the current iteration. After the All Pair SAA has been run with all the groups of isomers, our output file contains the BCD between each and every pair of isomers such that $2 \leq BCD \leq 2B$, where B is the maximum number of bonds per isomer we break. We have a lower limit of 2, since we break atleast one bond per isomer. For this discussion, we continue with the example of $M = 9$ from Section 4.1, where $N = 3$, and $d = 2$. We know our list contains the following isomer groups: (0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8), (0, 5, 7), (1, 3, 8), (2, 4, 6), (0, 4, 7), (1, 5, 6) and (2, 3, 8). We run the improved All Pairs SAA algorithm with each of these groups; i.e. iteration 1 will find the all pairs BCD between isomers 0, 1, and 2. Iteration 2 will find the all pairs BCD between isomers 3, 4 and 5, and so forth. At the end of 12 iterations, our output file will have bonds count distances between all 9 isomers $\{I_0, I_1 \dots I_8\}$.

Note that this approach involves no preprocessing and does not need any storage space on the disk. It is only constrained by the amount of isomers that can be processed in memory at one time.

However, **Step b** (in Figure 4.14 and Figure 4.15) is repeated more than once for each isomer. Each isomer is part of $\dfrac{N^d - 1}{N - 1}$ groups. Thus, though the output is the same, data is generated to the order of (M/N) times. To avoid this redundancy and make our process more efficient we employ our next approach.

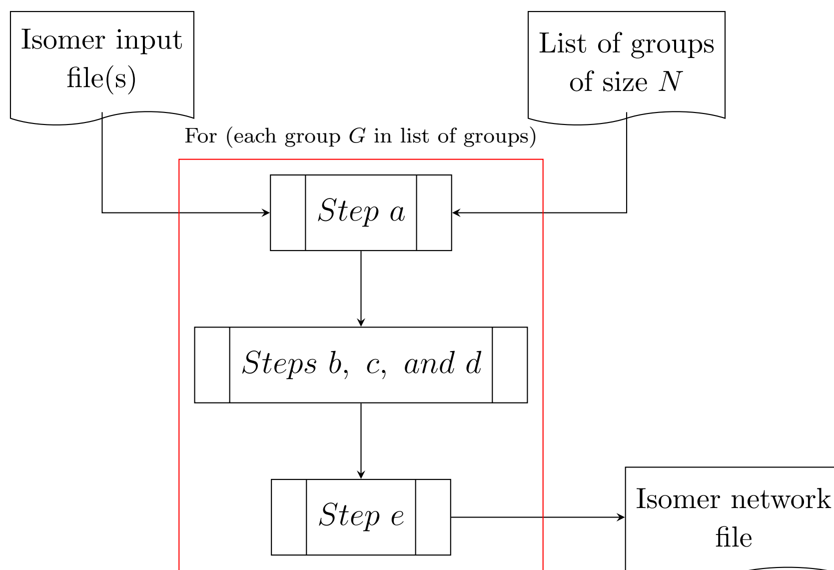


Figure 4.15: Flow chart for in-memory approach

4.6.2 File-based approach

All Pairs SAA [1] involves 1)generating a dnName after every bond is broken (up to B bonds) for each isomer 2) sorting dnNames corresponding to an individual isomer 3)merging dnNames of all isomers and 4) checking for duplicates. If two isomers I_x and I_y have the same dnName (obtained after breaking b_x and b_y number of bonds for I_x and I_y respectively), their Nauty names [12, 13] are generated. These Nauty names are then compared. If the Nauty names are also equal then $BCD(I_x, I_y) = b_x + b_y$.

The file-based approach divides the SAA mentioned above and in Figure 4.14 into two steps as shown in the flowchart in Figure 4.16 . In Step 1, we generate and store canonical names on disk for all the M isomers. In Step 2, we compute All Pairs BCD between isomers.

We describe two variations in Sections 4.6.2.1 and 4.6.2.2. The distinction between them involves comparing the efficiency of using dnNames versus Nauty names.

We briefly review how the dnName is generated (Figure 4.17). Consider an example of the CH_3O molecule in Figure 4.17(a). To obtain the dnName, we first label each atom using its symbol and degree (Figure 4.17(b)). We then add to each atom's name, the symbol and

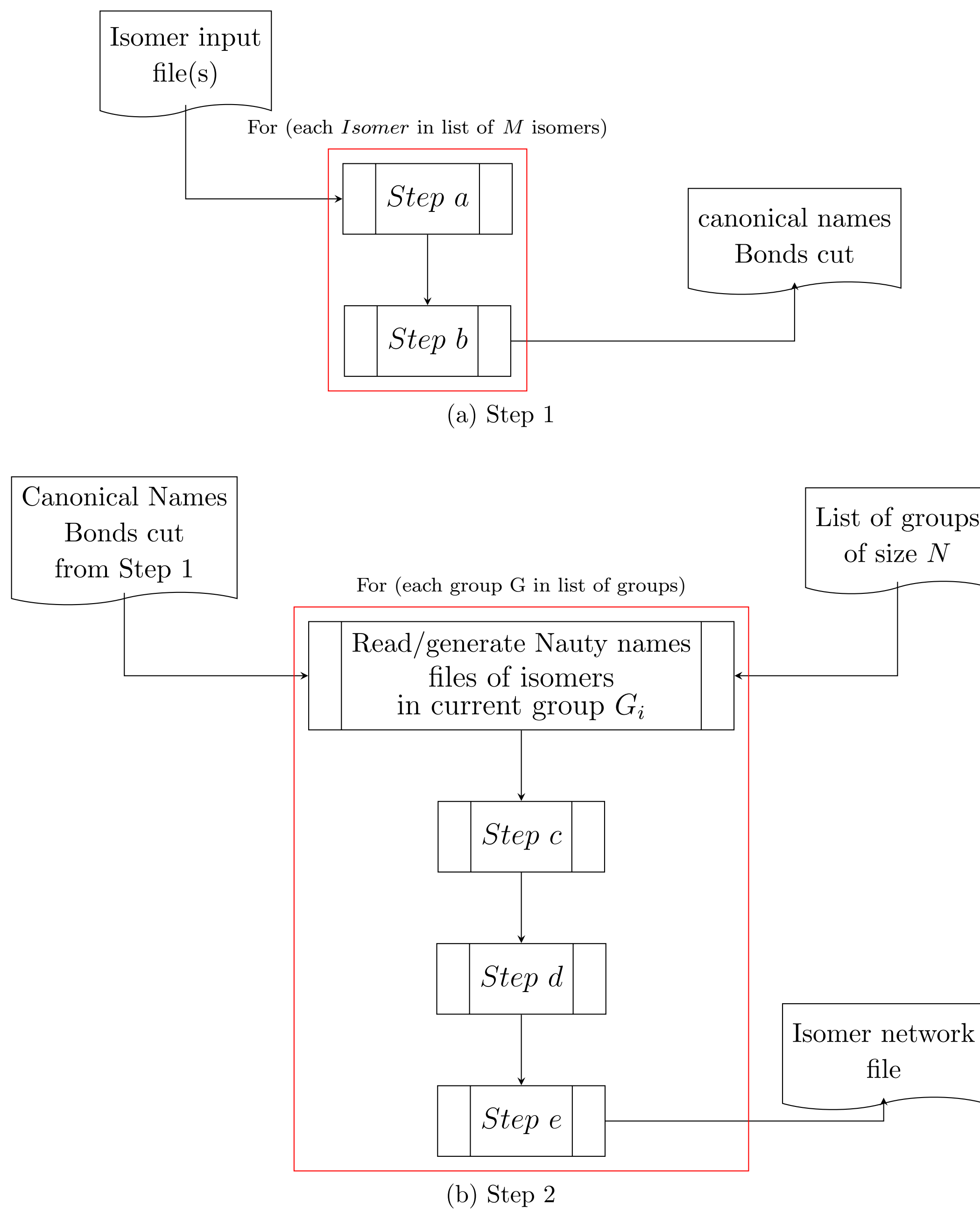


Figure 4.16: Flow chart for file-based approach

degree of its neighbors lexicographically (Figure 4.17(c)). Once each atom is named, we lexicographically sort the atom names to create the name for the molecule. More details on dnName generation can be found in [1].

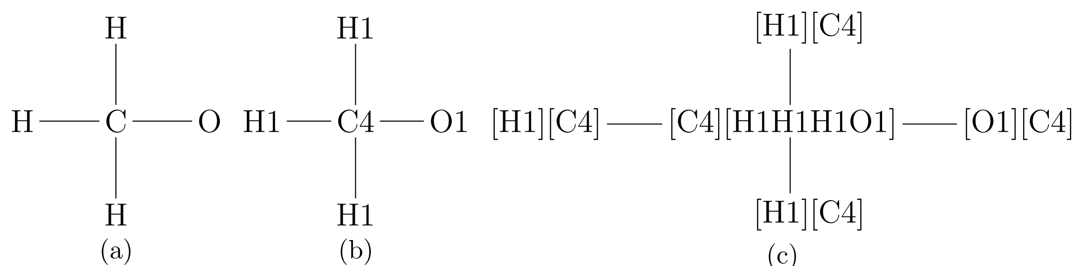


Figure 4.17: Degree Neighborhood Canonical Labeling

We obtain the Nauty name by calling the C program Nauty [12, 13].

The dnName generated for the CH_3O molecule is

[[C4][H1H1H1O1]][[H1][C4]][[H1][C4]][[H1][C4]][[O1][C4]]

The Nauty name generated is

C04H01H01H01O01

When we cut one bond in the molecule, say the C - O bond, the dnName generated is

[[C3][H1H1H1]][[H1][C3]][[H1][C3]][[H1][C3]][[O0]]

The Nauty name generated for the same input is

C03H01H01H01 O00

The dnName for a molecule is generated within the All Pairs SAA algorithm itself. Generating Nauty names requires considerable auxilliary information about the compound other that the dnName does not. It also has the additional overhead of calling the C program Nauty. This makes the dnName generation faster. We now discuss the two file-based approaches.

4.6.2.1 Store dnNames and compute Nauty names as needed

Step 1: We generate dnNames by breaking up to B bonds for each isomer I_x $(0 \leq x < M)$. B is the maximum number of bonds we break per isomer, 4 in our case. We store all the dnNames in a file. Each isomer has a seperate file that contains all the dnNames along with the number of bonds broken to arrive at that particular dnName. This is done as an independent process, run for all M isomers. Considering $M = 9$, at the end of Step 1, we will have nine files. $\{IsomerDN0, IsomerDN1 \ldots IsomerDN8\}$. Each file contains a list of dnNames and the number of bonds broken to arrive at that paricular dnName for the corresponding isomer.

Step 2: We retrieve the dnName files associated with the specific isomers in a group. We then compare these dnNames. When $(dnName_x == dnName_y)$, we conclude $BCD(I_x, I_y) = BondsCut_x + BondsCut_y$. We repeat Step 2 for our entire list of isomer groups. In our example, iteration 1 will retreive files $IsomerDN0$, $IsomerDN1$, and $IsomerDN2$. We find the dnNames that are equal for these three isomers, and then find the BCD amongst them. This is appended/written to the output file in the same format as described in Approach 1 (Section 4.6.1). Iteration 2 will do the same for isomers 3, 4 and 5, and so forth. At the end of 12 iterations, our output file will have bonds count distances between all 9 isomers $\{I_0, I_1 \ldots I_8\}$.

However comparing only dnNames is typically only 98% accurate. We still need to compare the Nauty names for 100% accuracy. This can be done two ways:

1. Retrieve the entire compound information only for those isomers I_x and I_y whose dnNames matched; i.e., run the entire all pair SAA algorithm for isomers I_x and I_y in order to generate the Nauty names of I_x and I_y which can then be compared. This is obviously not a very efficient solution, since it involves repeating the entire processing done in Step 1 above for isomers I_x and I_y .

2. Store auxiliary information needed to generate Nauty names along with the dnNames in the file. This will enable us to have all the pertinent information needed to generate the Nauty names once dnNames of two isomers are found to be similar. However this requires considerable amount of extra disk space (This becomes evident in the Results section of this chapter). The processing time needed to retreive the extra data from the file and generate the Nauty names from this information is also greater than the time it would take to generate the Nauty names in the first place as part of Step 1. This brings us to our next approach.

4.6.2.2 Store Nauty names

Step 1: We repeat Step 1 described in Section 4.6.2.1, except we generate and store the Nauty names after every bond is broken for each isomer instead of the $dnNames$.

Step 2: To find the bond count distances between isomers, we retrieve the Nauty name files associated with the specific isomers in each group, and compare the Nauty names of these isomers. When $(Nautyname_x == Nautyname_y)$, $BCD(I_x, I_y) = BondsCut_x + BondsCut_y$. The output is written as in previous approaches. The results are a 100% accurate.

In this approach we calculate the Nauty names for the entire set of M isomers. This is more time consuming than computing only dnNames. However storing Nauty names take up less space on the disk (since Nauty names are considerably smaller than dnNames as seen from the example described at the beginning of Section 4.6.2). Step 2 is also faster and more efficient. Another benefit to this approach is that once the Nauty names are stored in Step 1, Step 2 requires absolutely no access to the All Pairs SAA algorithm. It can be run as a stand alone program that returns the BCD between given isomer numbers.

4.6.3 Discussion

Step 1 of both the file-based approaches is not constrained by the number of isomers that can be processed together in memory at one time. This is because isomers are processed one at a time to store dnNames/ Nauty names. We iterate through Step 1 M times. It is however constrained by the amount of disk space available. Approach described in 4.6.2.2 is more efficient in this regard. The string size for each of the Nauty names is considerably smaller, and hence takes up less space on disk.

The value of N in Step 2, can be larger than it would be for the approach in Section 4.6.1. Most of the processing (including cutting bonds, and generating names) is done in Step 1. Step 2 only involves the process of generating bond count distances. This is simplified (since it is essentially comparing strings) and does not utilize as much intermediate memory as approach described in Section 4.6.1. Once we have the files from Step 1, Step 2 is platform independent and not constrained by any software/algorithmic needs. A simple program that can process files, and compare strings will suffice. This will give us immense flexibility, since the set up for running All Pair SAA is fairly complex.

Approach in Section 4.6.1 works best in situations where we do not have sufficient disk space.

4.7 Results

This section shows the results of running the SAA algorithm with both the approaches and the steps within them as described in Section 4.6. All experiments were carried out on a computer running Microsoft Windows Professional 7 with a 2.67 GHz Intel Core i7 CPU, 4GB of RAM, and around 100 GB free disk space. The algorithm to generate the groups of isomer networks was developed in C++.

We used the `Nauty` chemical graph canonical naming algorithm to test for isomorphism. We tested the code on multiple sets of isomers from the GDB database[5], including Nicotine isomers, Phenmetrazine isomers, and Tyrosine isomers.

Our data consisted of 529 isomers, where $N = 23$, and $d = 2$. There were a total of 552 isomer groups (each of size 23). We chose $N = 23$, since experiments showed we could process up to 25 isomers at one time (in-memory approach) in the laptop being used. 23 is the largest prime less than 25.

4.7.1 In-memory approach

We list the results of experiments run based on the approach described in Section 4.6.1 (also referred to as In-memory approach). We simply run the improved All Pairs SAA and find the BCD between 529 isomers (in batches of 23), where we cut up to 3 or 4 bonds per isomer. These results are listed in Table 4.12, Table 4.13, and Table 4.14.

Table 4.12: Results for Nicotine isomer

#Bonds cut	Time taken
up to 1	1.57 min
up to 2	6.85 min
up to 3	34.30 min
up to 4	7.56 hours

Table 4.13: Results for Phenmetrazine isomer

#Bonds cut	Time taken
up to 1	1.32 minutes
up to 2	6.10 minutes
up to 3	1.24 hours
up to 4	15.41 hours

4.7.2 File-based approach

In this section we list the results from our file-based approach in Section 4.6.2 (also referred to as Approach 2). Step 1 of both variations (Sections 4.6.2.1 and 4.6.2.2 - Approach 2a and Approach 2b) involve writing data to disk.

Table 4.14: Results for Tyrosine isomer

#Bonds cut	Time taken
up to 1	1.29 minutes
up to 2	5.98 minutes
up to 3	28.01 minutes

Approach 2a: Table 4.15 shows results when we write only the dnNames (Method 1) for **529 isomers** and also the estimated results the when we write dnNames *and* the auxiliary information needed to ensure 100% accuracy (Method 2). We ran experiments with a data set of 69 isomers and multiplied the results by a factor of 23/3 to extrapolate it to 529 isomers.

Table 4.15: Time and space required to write to file in **Step 1** (dnNames)

	Approach 2a			
	Method 1		**Method 2 (projected results)**	
	Size of file with only dnName	Time to write to file	Size of file with aux data included	Time to write to file
Nicotine	1.19 GB	38.29 min	1.79 GB	40.55 min
Phenmetrazine	2.21 GB	1.25 hours	2.85 GB	1.26 hrs
Tyrosine	910 MB	34.57 min	1.15 GB	36.25 min

Table 4.16 lists results for the experiments run on both approaches described under Step 2 of Section 4.6.2.1. In Method 1 we only compare the dnNames of all 529 isomers (in 552 groups of 23 isomers each). In order to get 100% accurate results we still need to 1) process compounds whose dnNames are similar, or 2) Read and process auxiliary information stored in Step 1.

To obtain results for Method 2, we ran experiments with 23 isomers and multiplied it by a factor of 552 to get results for calculating the all pairs BCD between all 23 isomers.

We notice under Method 2 of Table 4.16 that processing compounds is very time consuming. This is because when dnNames of two isomers are equal, we need to reload the entire

.mol for both isomers to generate their Nauty names, and find the BCD between them. Thus, isomer(compound) information is loaded multiple times for the same compound for multiple dnNames. For e.g., when we compute all pairs BCD between only 23 isomers of Nicotine, around 77023 pairs of dnNames to be compared. Thus this process becomes very inefficient. For this reason, we do not consider Method 2 to be a viable approach.

Table 4.16: Time and space required to process data from file in **Step 2** (dnNames)

	Approach 2a		
	Method 1	**Method 2 (projected results)**	
	Compare only dnNames	Process compounds	Read aux data from file
Nicotine	14.01 min	542 hrs	4.12 hrs
Phenmetrazine	31.20 min	744 hrs	5.21 hrs
Tyrosine	12.20 min	414 hrs	3.88 hrs

Approach 2b: Table 4.17 shows the amount of disk space needed and the time taken to generate and store Nauty names of all 529 isomers (under Step 1) and the time taken to generate all-pair BCD from Nauty names (under Step 2). Though Nauty names take longer to generate (49.44 minutes vs. 38.29 minutes for 529 Nicotine isomers, when breaking up to 4 bonds), they require less space on disk (410 MB vs. 1.19 GB). As explained at the beginning of Section 4.6.2, the string size for Nauty names is smaller. Hence sorting and processing times are faster, leading to less time taken to generate all pair BCD.

Table 4.17: Approach 2b (Nauty names)

	Approach 2b		
	Step 1		**Step 2**
	Size of file with Nauty name	Time to write to file	Time to find BCD
Nicotine	410 MB	49.44 min	8.25 min
Phenmetrazine	776 MB	1.71 hrs	32.6 min
Tyrosine	263 MB	40.96 min	5.18 min

We infer from the experimental results that the the best approach (when there is insufficient memory to process all M isomers) would be to generate Nauty names and store them on disk. We can then group isomers where the group size depends on amount of isomers that can be processed in memory at one time.

CHAPTER 5

ISOMER NETWORK GENERATION AND GRAPH ANALYTICS

As discussed in the previous chapters, the amount of compute time and space required to build the isomer network (even as a 2 step process) makes it infeasible to accomplish on a standalone machine. In this chapter, we use cloud computing to generate the Nauty names for a million Nicotine isomers, and subsequently their isomer network. To achieve this, we applied for and received cloud research credit scholarships from both Amazon[2] and Microsoft[3] worth a total of $24,500 ($20,000 from Microsoft and $4,500 from Amazon).

In Section 5.1, we describe the process of isomer network generation for 1,050,125 Nicotine isomers (the number of valid Nicotine isomers from the 1,050,219 we started out with). In Section 5.2 and Section 5.3 we show our results from analyzing an order-4 network of approximately 500,000 and 1,000,000 isomers. An order-4 network is one where we cut up to 4 bonds per isomer. In Section 5.4 we provide a description of the graph analytical metrics used in Section 5.2 and Section 5.3.

5.1 Generating the isomer network

Recall that isomer network generation consists of two steps.

1. Generate and store Nauty names for each isomer along with the number of bonds cut to arrive at each nauty name for every isomer. (One file is generated per isomer that contains a list of Nauty names and corresponding bonds cut)

2. Use the files generated in Step 1 to generate the isomer network.

5.1.1 Genarating and storing Nauty names for each isomer

We generated Nauty names using Amazon Web Services (AWS)[2]. The following steps were used to run our application on AWS. Technical AWS-specific information presented here is reproduced from the AWS website. [2, 27, 28]

1. *Connect to AWS account and choose an Amazon Machine Image (AMI):* An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch our instance. We chose the Amazon Linux AMI 2017.03.0 (HVM), SSD Volume Type. Based on documentation regarding AMIs, this was best suited to run our application. The default image includes AWS command line tools, Python, Ruby, Perl, and Java.

2. *Choose Instance type:* Instances are virtual servers that can run applications. Amazon Elastic Cloud Compute (EC2) provides a wide selection of instance types optimized to fit different use-cases. These instances have varying combinations of CPU, memory, storage, and networking capacity, that provide the flexibility to choose the appropriate mix of resources needed. The instance types provided by AWS are listed below.

   - *General purpose instances* provide a balance of compute, memory, and network resources.
   - *Compute optimized instances* have a higher ratio of vCPUs to memory than other families and are recommended for running CPU-bound applications. A vCPU in an AWS environment actually represents half a physical core.
   - *GPU instances* provide graphics processing units (GPUs) along with high CPU and network performance for applications benefiting from highly parallelized processing, including 3D graphics, HPC, rendering, and media processing applications.
   - *Memory optimized instances* have the lowest cost per GB of RAM among Amazon EC2 instance types and are recommend for memory intensive applications.

   In order to choose the instance type, we conducted preliminary experiments between general, compute-optimized and memory-optimized instances (by running sample data sets on various instance types). We then picked the general purpose instance type as the most suitable to our application.

3. *Choose instance:* Once we chose the instance type, we conducted experiments on different instances (M3 and M4 series). We picked the m4.2xlarge model as the most time and cost efficient instance for our purposes. (M4 instances are the latest generation of general purpose instances) The m4.2xlarge model has 8vCPUs, and 32 GB of memory.

4. *Launch instance and connect using ssh:* After accepting default setting for all other configurations (like security), we launched the instance. Once the instance is launched, we generated a key pair (for the first time) to use to launch putty and file transfer sessions. The same key pair can be used during subsequent instances. We use the IP address of the instance to ssh into the machine.

5. *Set up environment on the machine:* Once we are logged into the machine, we had to install the software required to run our application. In our case Java was already installed, but in order to zip and unzip files to and from the AWS instance, we had to install zip and unzip.

6. *Upload input files and Java application from local machine:* We used Filezilla[29] to upload input files and Java application.

7. *Run the Java application* to generate Nauty names from the command line.

8. *Move result set to Amazon S3 storage:* We created a bucket in the same region as EC2 instance (We used the US West - Oregon region as part of my scholarship). Moving data directly from EC2 instance to Amazon S3 is faster than downloading it to our local machine, since the EC2 instance and S3 bucket are in the same physical region.

5.1.2 Experience and Results with AWS

We now describe the experience and resources utilized to generate the 1,050,125 files containing Nauty names and number of bonds cut for each isomer.

We ran the application in batches. Each batch processed 5,000 .mol (input) files. Recall that a .mol file (discussed in Chapter 3) contains the structure of each Nicotine isomer

in a fixed format read by our program. Each .mol file has a size of 3 KB. It took 15 minutes to transfer input files (15,000 KB of data) from local PC to the AWS EC2 instance. Generating Nauty names (along with number of bonds cut to arrive at each Nauty name) for each set of 5,000 input files took approximately 20 minutes. The Nauty name files for 5,000 isomers requires 1.5 GB. There were approximately 2,500 Nauty names per isomer. This number varies depending on the symmetry present in each isomer. Time to move these files to Amazon S3 storage was around 15 minutes per 5,000 files. Since S3 does not support zipping and unzipping files, volume of data transferred for each iteration (batch) was 1.5 GB.

Thus each iteration of 5,000 files took about one hour. Some of these steps could be done in parallel on the same instance, and also with multiple EC2 instances.

We were able to process 50,000 isomers each day (This is in contrast to 10,000 isomers in the same time period on my desktop.)

Total cost incurred to generate Nauty names of 1,050,125 isomers and store them in AWS was $450. Total size of NautyName files was around 325 GB. Note that this is the size of the final file(s), which is considerable less than the intermediate space required to generate this data.

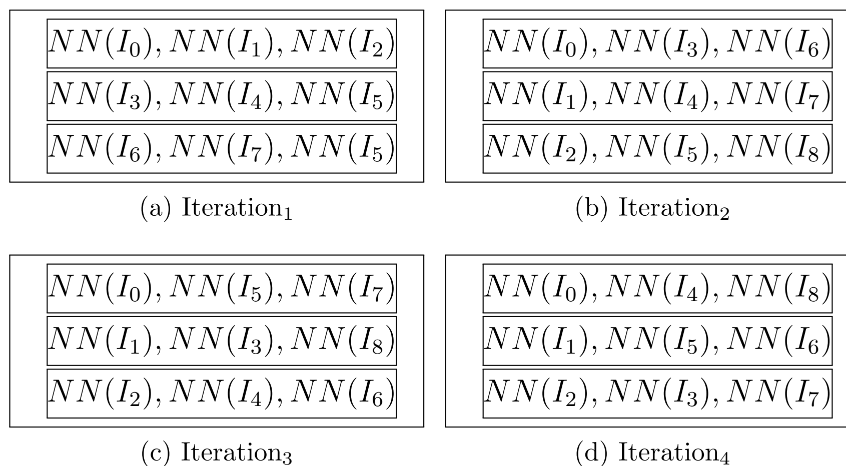
5.1.3 Generating isomer network from Nauty names

This section describes generating the isomer network. Since it was still not feasible to generate the isomer network of all 1,050,125 isomers at the same time, we used the grouping algorithm described in Chapter 4 to divide the 1,050,125 isomers into groups.

The group size (number of isomers to run in memory at the same time) we chose is 1031. $1031^2 = 1,062,961$, which is $\geq$ 1,050,125. We use -1 for all entries greater than 1,050,125. Each iteration of the algorithm mentioned in Chapter 4 was stored in a separate file (to enable parallel generation of isomer networks).

We first consider the example of applying the grouping technique for 9 isomers ($I_0, I_1, \cdots, I_8$) where only 3 isomers can fit into memory at one time. (This example is discussed in

detail in Section 4.1). The grouping of isomers in each iteration is shown in Figure 5.1.



(a) Iteration_1 (b) Iteration_2 (c) Iteration_3 (d) Iteration_4

Figure 5.1: Iterations using the grouping technique

$Iteration_1$ is the output of Step 0 in Figure 4.4, while $Iteration_2$, $Iteration_3$, and $Iteration_4$ are the outputs of Step 1 in Figure 4.4. (Please refer to Section 4.1 for specifics on how we arrive at isomers in each group). We make the following observations (for $N = 3$).

1. We have four (N + 1) iterations.
2. Every isomer is in a group with every other isomer in every iteration.
3. Every iteration consists of 3 (N) groups.
4. Each group has 3 (N) isomers.

We now extend this to grouping 1031^2 isomers ($N = 1031$). Since we stored the output of each iteration in a separate file, there were 1032 files. (1 file for the row wise grouping of isomers and 1031 for column wise grouping). Each file had 1031 groups with 1031 isomers in each group. For each file (iteration) all 1,050,125 Nauty name files are accessed. Thus we access the 1,050,125 Nauty name files (325 GB) 1032 times.

Each of our Nauty name files could be accessed from AWS S3 by the EC2 instance at a rate of 15 files/sec. This rate would not have been feasible when we needed to access

all 1,050,125 files 1032 times. Also there was not enough local disk space to download entire 325 GB to an EC2 instance. For this reason we chose to generate isomer networks in Microsoft Azure. MS Azure provides virtual machines similar to AWS EC2 instances. It also provides the capability to mount local datadrives to a virtual machine (The data drives are not persistent storage and are available for only as long as an instance is running.)

The following steps were used to configure a virtual machine on MS Azure and launching our application to generate the isomer network. All technical Azure-specific information presented here is reproduced from the Microsoft Azure website. [3, 30, 31])

1. *Login to Azure portal and choose type of application:* We chose compute from the group of Apps, and Ubuntu Server 16.06 LTS.

2. *Create virtual machine:* Azure links every virtual machine to a deployment model. (The deployment models provided by Azure are “resource manager” where Azure manages our resources, or “classic” where we have to manage our resources). For our application, we chose resource manager as our deployment model. Every virtual machine is also connected to a Resource group (which is a collection of resources that share the same lifecycle, permissions, and policies). We created a resource group the first time, and used the same one for every subsequent instance.

3. *Pick the region in which to deploy the virtual machine and its configuration:* We were limited to some regions, and the number of cores we can use in each region (and the total number of cores we can use at a single time). The machines and configurations available in each region also vary. We chose US West2 and US East regions since they had the configurations we wanted to use. The configuration we used was 16 cores, 112 GB memory, 32 data disks (number of data disks that can be mounted) with 50,000 Max IOPS. During configuration, we also added a data disk of 512 GB. Each virtual machine has to have it’s own data drive. So we had transfer all 1,050,125 files to every data disk individually. In order to get maximum benefit from this effort, we picked

a machine with 16 cores, and high RAM, so we could run multiple instances of our program. Other options were 8 and 4 cores. (With 16 cores, our CPU utilization is an average of 65%.) This was the highest configuration allowed under my scholarship.

4. *Launch the virtual machine and connect using ssh:* Once the virtual machine is launched (takes about 20 minutes), we ssh into the machine (using a password).

5. *Set up environment on the machine:* Once we are logged into the machine, we need to mount the data disk (added during creating the virtual machine) and install the software needed to run our application. Since Azure virtual machines do not come with Java by default we installed Java, Zip, and Unzip.

6. *Upload input files and Java application from local machine:* We used Filezilla to upload input files, Java application, and the list of groups. We uploaded files in zipped form, and unzipped them in the data disk.

7. *Run the Java application to generate the isomer network from the command line*

8. *Move result set back to local PC:* The result set is a group of files containing the network (Files contain entries of the form $(I_1\ I_2\ BCD(I_1, I_2))$.

5.1.4 Experience and Results with MS Azure

We now describe the experience and resources utilized to generate the isomer network for 1,050,125 isomers.

Since it was not feasible to generate the network on AWS, we had to download all 325 GB from Amazon S3 to a local machine before we could upload it to MS Azure. Since the S3 portal does not provide the capability of zipping files, we logged into another EC2 instance, downloaded the files (from S3 to the EC2 instance). We then zipped these files in the EC2 instance, and downloaded the zipped files onto the local pc. The total size of all zipped files was 65 GB.

Uploading input files and Java application from local machine to a virtual machine (after it has been configured) involves the following steps.

1. Copy all zipped input files from local machine to virtual machine. Given the volume of data, this took around 12 hours (All input was stored in 100 zip files)

2. Unzip all 100 files and move them to the correct input folder for the application (Took around 2 hours)

3. Simultaneously load Java application and files containing groups of isomers.

Thus, the set up of each virtual machine took around 14 hours. We set up three virtual machines. The above process had to be repeated three times, since Azure does not support data disks being shared between multiple virtual machines.

Each file (set of 1031 groups) took around 2.5 hours to be processed. In each virtual machine, 10 instances of the Java program (i.e., 10 files) could be executed at the same time. Since we set up three instances, we processed 30 files at a time every 2.5 hours. The total cost involved in generating the isomer network of 1,050,125 isomers is $700.

5.2 Analytics for graph with around 500K vertices

This section contains the metrics run on the isomer network of approximately 500,000 isomers of Nicotine. Though we report the metrics and what each metric captures, understanding and rating the actual significance of these metrics for isomer network requires significant domain knowledge and further collaboration with scientists at the Chemical Space Project[4].

We refer to the graph generated from the isomer network as G_{500K}. We start with the simplest metrics of the graph,

$vcount(G_{500K}) = 500786$ and $ecount(G_{500K}) = 4228958$.

Recall that the graph is weighted with edge-weights equal to 2 or 4 depending on the bond count distance between the isomers that form the vertices that make edge.

5.2.1 Centrality

Centrality indicates how important a given vertex or an edge is in a graph.

1. **Vertex centrality**

   Figure 5.2 shows the frequency of degrees of vertices. We see that the maximum degree of a vertex in the network is 202. Table 5.1 shows the top 25 vertices sorted on the basis of vertex centrality in G_{500K}.

Table 5.1: Top 25 vertices of G_{500K} sorted on vertex centrality

Vertex	**Degree**
98851	202
98791	202
41962	192
41957	192
11188	192
11193	192
266154	188
231170	188
231154	188
266170	188
41960	181
98780	181
11191	181
98840	180
42001	178
11226	178
220371	177
255371	177
255383	176
220383	176
41959	173
11246	173
42023	173
11190	173
11257	170

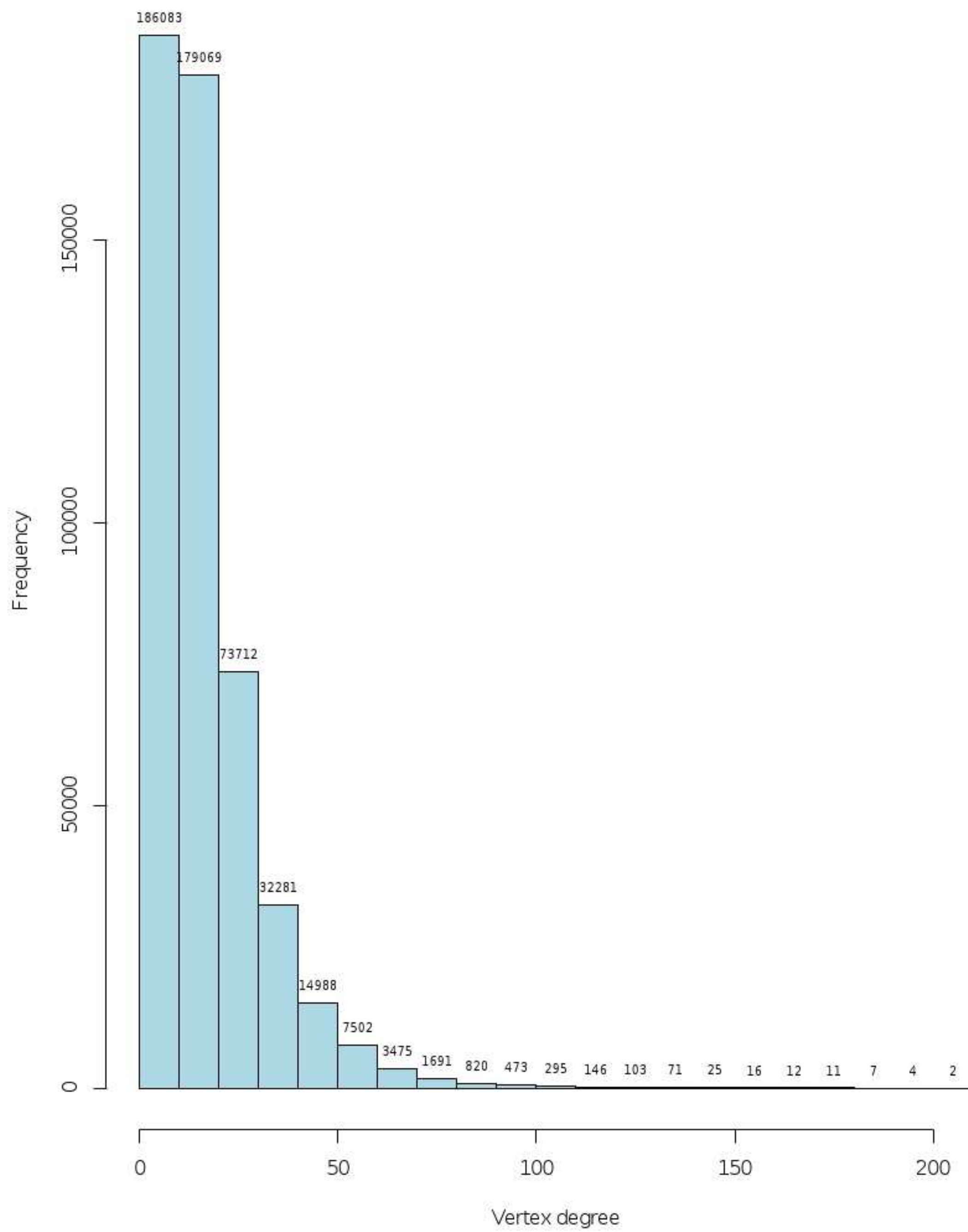


Figure 5.2: Frequency of degree of vertices in G_{500K}

2. **Closeness centrality**

   Since our graph is large, we calculate estimated closeness centrality with a cutoff of 10. Table 5.2 shows the top 25 vertices sorted on the basis of closeness centrality in G_{500K}.

Table 5.2: Top 25 vertices of G_{500K} sorted on estimated closeness centrality with cutoff = 10.

Vertex	**Closeness cent.**
2650	4.121349e-12
220371	4.119699e-12
2663	4.119419e-12
220383	4.118986e-12
98851	4.118281e-12
255371	4.117975e-12
11945	4.117041e-12
255383	4.116455e-12
2267	4.115463e-12
2260	4.115463e-12
98783	4.115319e-12
266606	4.114895e-12
3200	4.113047e-12
3397	4.112971e-12
98791	4.112759e-12
2416	4.112403e-12
255424	4.111988e-12
166352	4.111692e-12
2404	4.110456e-12
166400	4.110346e-12
169904	4.110016e-12
257078	4.109686e-12
2452	4.109593e-12
2397	4.109390e-12
11967	4.109179e-12

3. **Betweenness centrality**

   We also calculate the estimated betweenness centrality with a cutoff of 10. Table 5.3 shows the top 25 vertices sorted on the basis of betweenness centrality in G_{500K}.

Table 5.3: Top 25 vertices of G_{500K} sorted on estimated betweenness centrality with cutoff = 10.

Vertex	**Betweenness cent.**
31275	115331.19
42847	111780.09
16956	111025.25
266606	97004.33
72617	91237.21
42897	90775.18
23053	88927.75
30489	85702.95
1512	82474.67
16957	80934.92
42851	79535.66
98783	78760.39
16959	75842.07
57170	73479.89
260442	72329.64
1615	72326.93
98851	72026.51
220383	69580.37
41957	69287.83
11945	68978.08
294694	67440.13
42839	67120.97
43486	67011.43
160316	65716.73
2650	65518.10

4. **Eigenvector centrality**

Table 5.4 shows the top 25 vertices sorted on the basis of eigenvector centrality for G_{500K}.

Table 5.4: Top 25 vertices of G_{500K} sorted on eigenvector centrality

Vertex	**Eigenvector centrality**
22695	1.0000000
43485	0.9922228
22685	0.9919571
42851	0.9828233
42885	0.9796251
1928	0.9769588
111855	0.9722704
23005	0.9717112
31275	0.9702614
1926	0.9698272
42837	0.9680700
43053	0.9647908
42836	0.9642023
72617	0.9641326
4645	0.9625898
22949	0.9620964
22605	0.9592186
1512	0.9585352
43088	0.9582718
42847	0.9581261
42886	0.9576505
43229	0.9570072
43486	0.9561863
22690	0.9548654
22682	0.9545745

Observations on centrality:

There are three vertices at the intersection of top 100 of all four measures. These are vertices 42836, 42839 and 42847. There are 27 vertices in the intersection of the top 500 of all four measures. (These are vertices 42839, 42836, 42847, 42875, 42886, 22948, 43092, 23009,

43103, 42851, 31275, 72617, 42837, 43486, 42897, 42834, 1512, 42885, 2710, 2697, 43040, 43046, 23005, 43053, 4632, 22762 and 22935.)

5.2.2 Centralization

Centralization is a method for creating a graph-level centralization measure from the centrality scores of the vertices. We measure how central the most central node in the network is in relation to all other nodes. Table 5.5 shows centralization measures computed for G_{500K}.

Table 5.5: Centralization measures computed for G_{500K}

Centrality Measure	**Value**	**Theoretical Max**
Degree	0.000369	250785115440
Eigen Vector	0.999737	500784

5.2.3 Network cohesion

Here are some metrics calculated to indicate how cohesive the network is.

transitivity(G_{500K}) = 0.1786286

count_maximal_cliques (G_{500K}) = 140,485,252 (Indicates that there are 140,485,252 maximal cliques)

clique_num(G_{500K}) = 61 (Indicates that the largest clique(s) is of size 61)

Figure 5.3 shows the frequency of $k - core$ subgraphs. We note that the maximum value for k is 86.

G_{500K} has 389 components. We see in Table 5.6 that G_{500K} has 1 giant component with 499,512 vertices.

5.2.4 Community detection

We list the results of community detection algorithms we ran on G_{500K} in Table 5.7 .

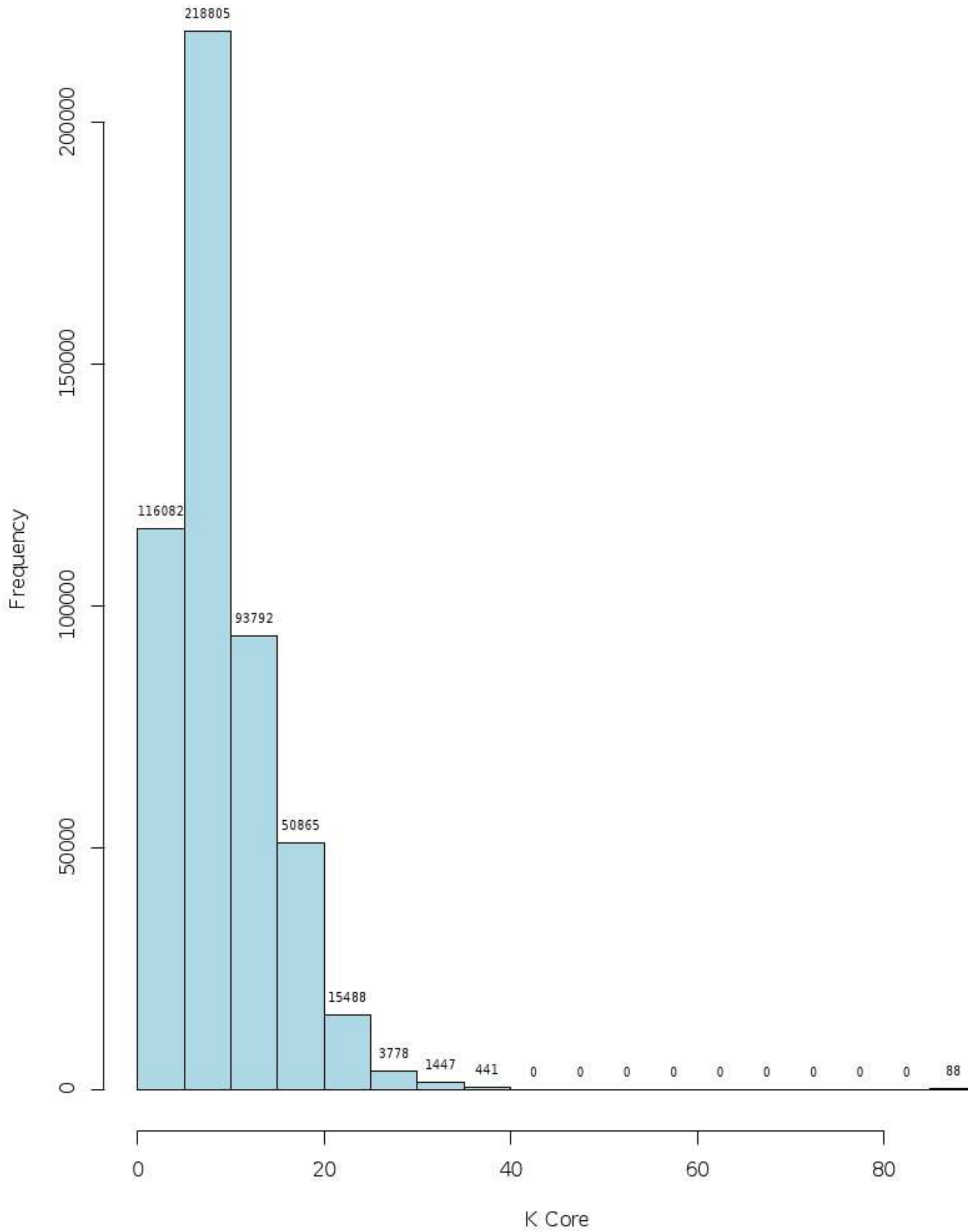


Figure 5.3: Frequency of cores for G_{500K}

Table 5.6: Number of components and their sizes in G_{500K}

2	3	4	5	6	7	8	9	15	16	20	21	48	49	64	499512
241	73	39	11	10	2	2	2	1	1	1	2	1	1	1	1

Table 5.7: Results of community detection algorithms run on G_{500K}

Algorithm	**Number of clusters**	**Size of largest cluster**	**Modularity**
Label propagation	13223	26446	0.71
Louvian	427	36201	0.87
Fast greedy	438	58853	0.85
Optimal	Not suited to large graphs		
Leading eigenvector	Does not work for weighted graphs		
Spinglass	Does not work for unconnected graphs		
Infomap	Suited to directed graphs		
Edge betweenness	did not complete execution in 8 hours		
Walktrap clustering	did not complete execution in 8 hours		

5.3 Analytics for graph with a million vertices

This section contains the metrics run on the isomer network of approximately 1,000,000 isomers of Nicotine. Though we report the metrics and what each metric captures, understanding and rating the actual significance of these metrics for isomer network requires significant domain knowledge and further collaboration with scientists at the Chemical Space Project[4].

We refer to the graph generated from the isomer network as G_{1M}. We start with the simplest metrics of the graph

$vcount(G_{1M}) = 1046670$ and $ecount(G_{1M}) = 7844263$.

Recall that the graph is weighted with edge-weights equal to 2 or 4 depending on the bond count distance between the isomers that form the vertices that make edge.

5.3.1 Centrality

Centrality indicates how important a given vertex or an edge is in a graph.

1. **Vertex centrality**

   Figure 5.4 shows the frequency of degrees of vertices. We see that the maximum degree of a vertex in the network is 205.

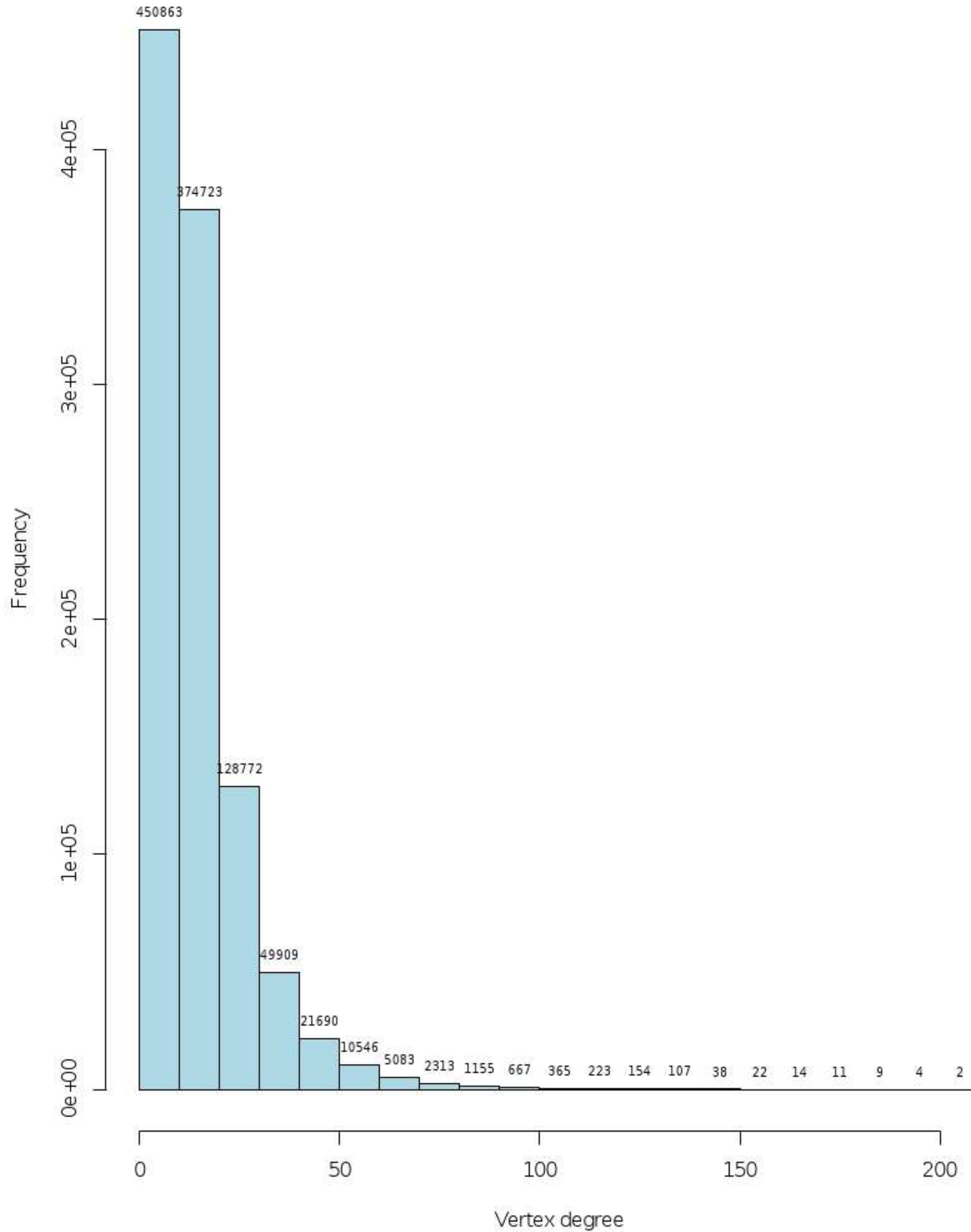


Figure 5.4: Frequency of degree of vertices in G_{1M}

Table 5.8 shows the top 25 vertices sorted on the basis of vertex centrality in G_{1M}.

Table 5.8: Top 25 vertices of G_{1M} sorted on vertex centrality

Vertex	**Degree**
98791	205
98851	205
11188	194
41957	194
41962	192
11193	192
266154	188
231170	188
231154	188
266170	188
220371	181
11191	181
41960	181
255371	181
98780	181
98840	180
42001	180
11226	180
220383	180
255383	180
42023	174
11246	174
42042	173
41959	173
11257	173

2. **Closeness centrality**

   Since our graph is large, we calculate estimated closeness centrality with a cutoff of 12. Table 5.9 shows the top 25 vertices sorted on the basis of closeness centrality in G_{1M}.

Table 5.9: Top 25 vertices of G_{1M} sorted on estimated closeness centrality with cutoff = 12.

Vertex	**Closeness cent.**
166400	9.539071e-13
266106	9.534987e-13
260514	9.531981e-13
266122	9.529937e-13
231606	9.526981e-13
220383	9.526649e-13
230426	9.524939e-13
255371	9.524816e-13
160316	9.523658e-13
169904	9.521019e-13
164651	9.517036e-13
2710	9.515984e-13
269135	9.515700e-13
267255	9.515273e-13
166428	9.511153e-13
166344	9.510982e-13
220371	9.510641e-13
98851	9.509023e-13
98791	9.508436e-13
11967	9.508067e-13
166396	9.507764e-13
166230	9.507745e-13
2663	9.505693e-13
260393	9.505560e-13
164609	9.505409e-13

3. **Betweenness centrality**

   We also calculate the estimated betweenness centrality with a cutoff of 12. Table 5.10 shows the top 25 vertices sorted on the basis of betweenness centrality in G_{1M}.

Table 5.10: Top 25 vertices of G_{1M} sorted on estimated betweenness centrality with cutoff = 12.

Vertex	**Betweenness cent.**
2710	564903.5
31275	485033.8
2663	481839.9
3397	463051.8
269135	456732.6
42839	454891.9
220383	451112.7
248967	430590.6
164651	418904.8
42847	416150.8
72617	413102.4
23053	369452.3
114048	367282.7
42834	366440.2
56714	361782.0
255371	356460.0
169342	352434.5
166396	350278.2
248966	349310.6
11962	347892.9
352420	343044.4
480740	342418.1
166280	340957.9
42046	339209.7
164573	337632.2

4. **Eigenvector centrality**

   Table 5.11 shows the top 25 vertices sorted on the basis of eigenvector centrality for G_{1M}.

Table 5.11: Top 25 vertices of G_{1M} sorted on eigenvector centrality

Vertex	**Eigenvector centrality**
1512	1.0000000
72617	0.9995556
42875	0.9983559
43229	0.9981192
22718	0.9977044
4643	0.9971177
22948	0.9961938
42836	0.9959577
22616	0.9958880
42839	0.9958450
22682	0.9958407
43053	0.9948911
42897	0.9948407
42847	0.9943386
43088	0.9940327
43092	0.9938852
43485	0.9935494
43103	0.9933022
42851	0.9930201
43046	0.9926835
22715	0.9926451
43226	0.9923880
22746	0.9923792
43102	0.9920568
42834	0.9918637

Observations on centrality:

There are 11 vertices in the intersection of the top 500 of all four measures. (These are vertices 42839, 42836, 42847, 23009, 31275, 72617, 43486, 42897, 42834, 2697 and 2710.) There are 26 vertices in the intersection of the top 1000 of all four measures. (These are vertics 42839, 42836, 42847, 42886, 23009, 70495, 21892, 31275, 72617, 42837, 43486, 42897,

42834, 1512, 2697, 2710, 42885, 22581, 4632, 22699, 22651, 2569, 2590, 22935, 4645 and 31432)

5.3.2 Centralization

Centralization is a method for creating a graph-level centralization measure from the centrality scores of the vertices. We measure how central the most central node in the network is in relation to all other nodes. Table 5.12 shows centralization measures computed for G_{1M}.

Table 5.12: Centralization measures computed for G_{1M}

Centrality Measure	**Value**	**Theoretical Max**
Degree	0.0001815388	$1.095515e + 12$
Eigen Vector	0.9998709	1046668

5.3.3 Network cohesion

Here are some metrics calculated to indicate how cohesive the network is.

transitivity(G_{1M}) = 0.157706

count_maximal_cliques (G_{1M}) = 142,948,282 (Indicates that there are142,948,282 maximal cliques)

clique_num(G_{1M}) = 61 (Indicates that the largest clique(s) is of size 61)

Figure 5.5 shows the frequency of $k - core$ subgraphs. We note that the maximum value for k is 88.

G_{1M} has 480 components. We see in Table 5.13 that G_{1M} has 1 giant component with 1,045,319 vertices.

5.3.4 Community detection

We list the results of community detection algorithms we ran on G_{1M} in Table 5.14 .

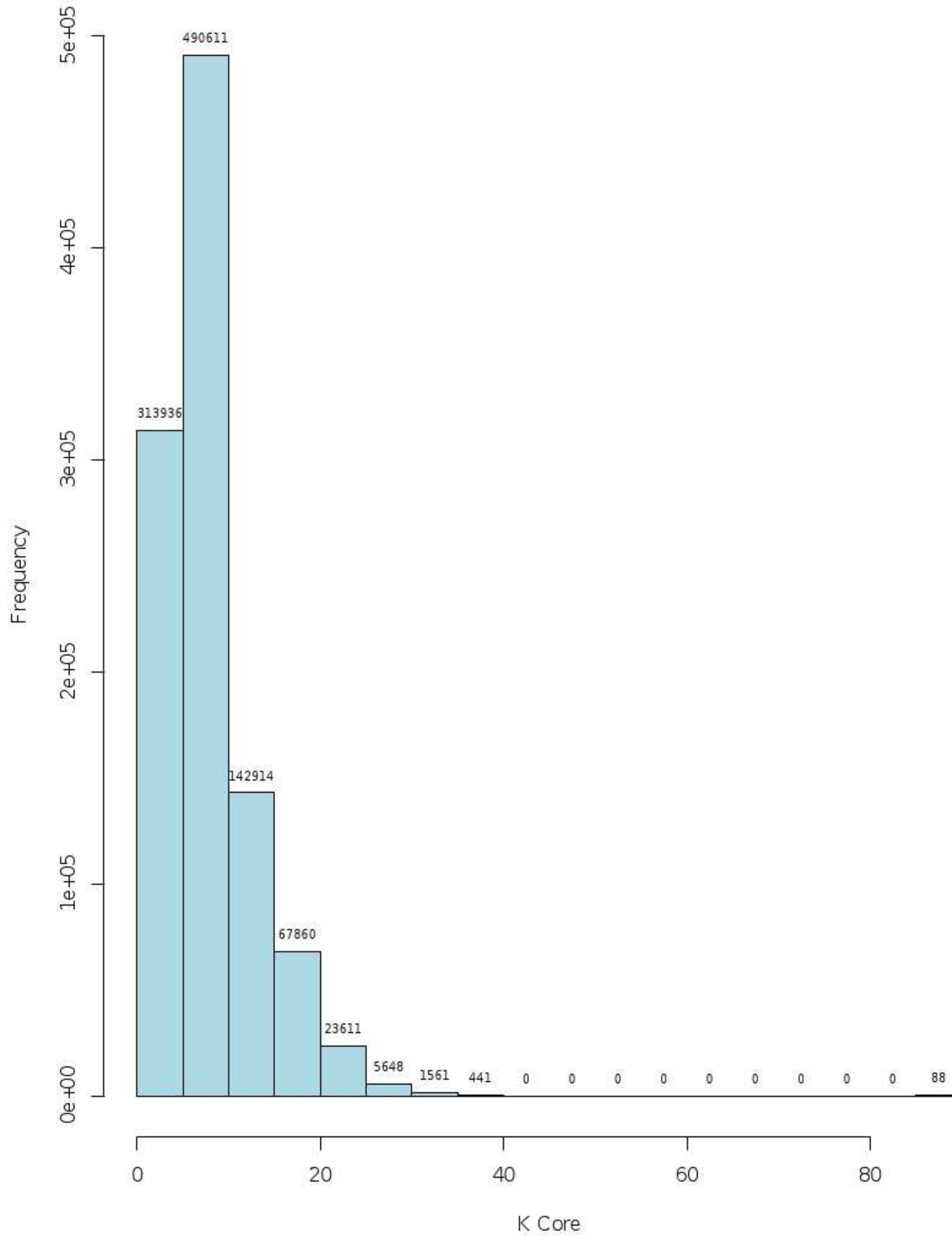


Figure 5.5: Frequency of cores

Table 5.13: Number of components and their sizes in G_{1M}

2	3	4	5	6	7	8	9	10	16	20	49	64	1045319
341	72	38	9	7	3	2	2	1	1	1	1	1	1

Table 5.14: Results of community detection algorithms run on G_{1M}

Algorithm	**Number of clusters**	**Size of largest cluster**	**Modularity**
Label propagation	10197	361864	0.74
Louvian	520	133756	0.88
Fast greedy	1359	365622	0.76
Optimal	Not suited to large graphs		
Leading eigenvector	Does not work for weighted graphs		
Spinglass	Does not work for unconnected graphs		
Infomap	Suited to directed graphs		
Edge betweenness	did not complete execution in 8 hours		
Walktrap clustering	did not complete execution in 8 hours		

5.4 Graph Analytics with an example graph

As mentioned in Chapter 1, network analytics techniques [6–11, 26] that have been developed in other research communities can be applied to chemical networks to yield insights about chemical space.

Figure 5.6 is a classic example of social network showing fifteenth-century Florentine marriages[32]. The vertices are the families in the oligarchy that ruled Florence at that time and the edges are the connections made through marriages. Historically, the Medici have been called the "godfathers of Renaissance". They accumulated immense power in the early fifeenth century even though they started with less wealth and political clout than other families in the oligarchy that ruled Florence at that time (other vertices in graph)[32]. The key to the Medici family's rise was found to be in their network structure. They strategically placed themselves in a position to make alliances with other families, and also be part of alliances other families may want to have with each other (as can be seen in Figure 5.6).

In this section, we explain the graph (social) analytic metrics that were applied to the isomer network (results of which were presented in Sections 5.2 and 5.3). We use a running

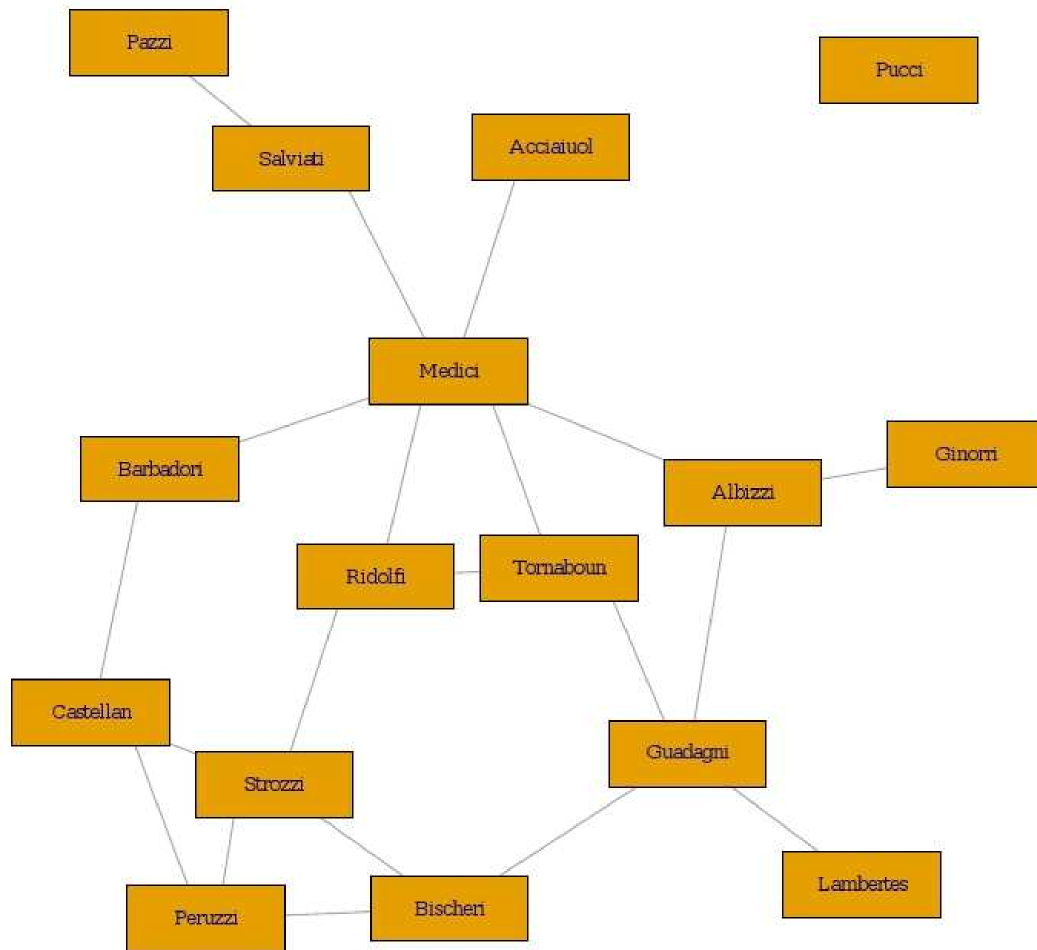


Figure 5.6: Graph showing fifteenth-century Florentine marriages[32]

example with graph (g) shown in Figure 5.7 which is the reproduced from the graph with named vertices in Figure 5.6 to describe the metrics.

Most of the material presented in this section is reproduced from [6, 11, 32]. We used the igraph package on R[6] to calculate these metrics and definitions presented in this section are reproduced from the igraph manual.

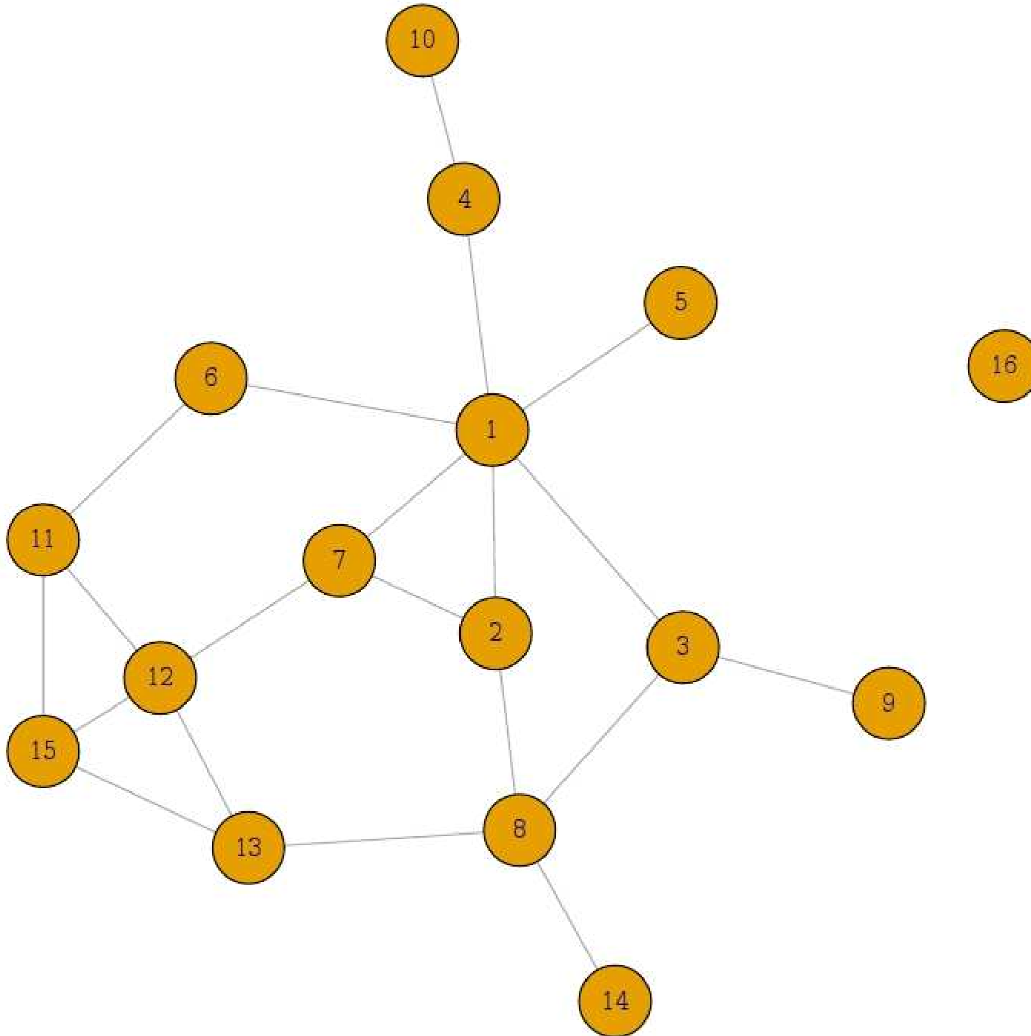


Figure 5.7: Example graph (g)

The simplest metric for a graph is the number of vertices and the number of edges. For our example, number of vertices and number of edges are $vcount(g) = 16$ and $ecount(g) = 20$ respectively.

5.4.1 Other metrics

In this section, we describe some metrics applicable to networks.

Definition *distances* calculates the length of all the shortest paths from or to the vertices in the network.

Definition The *shortest path*, or *geodesic* between two pair of vertices is a path with the minimal number of vertices.

By default `igraph` tries to select the fastest suitable algorithm to calculate the shortest path. If there are no weights, then a breadth-first search is used. If all weights are positive, then Dijkstras algorithm [33] is used.

Table 5.15: Table showing shortest distances between every pair of vertices in graph g.

	Vertices															
Vertices	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
1	0	1	1	1	1	1	1	2	2	2	2	2	3	3	3	∞
2	1	0	2	2	2	2	1	1	3	3	3	2	2	2	3	∞
3	1	2	0	2	2	2	2	1	1	3	3	3	2	2	3	∞
4	1	2	2	0	2	2	2	3	3	1	3	3	4	4	4	∞
5	1	2	2	2	0	2	2	3	3	3	3	3	4	4	4	∞
6	1	2	2	2	2	0	2	3	3	3	1	2	3	4	2	∞
7	1	1	2	2	2	2	0	2	3	3	2	1	2	3	2	∞
8	2	1	1	3	3	3	2	0	2	4	3	2	1	1	2	∞
9	2	3	1	3	3	3	3	2	0	4	4	4	3	3	4	∞
10	2	3	3	1	3	3	3	4	4	0	4	4	5	5	5	∞
11	2	3	3	3	3	1	2	3	4	4	0	1	2	4	1	∞
12	2	2	3	3	3	2	1	2	4	4	1	0	1	3	1	∞
13	3	2	2	4	4	3	2	1	3	5	2	1	0	2	1	∞
14	3	2	2	4	4	4	3	1	3	5	4	3	2	0	3	∞
15	3	3	3	4	4	2	2	2	4	5	1	1	1	3	0	∞
16	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	0

Table 5.15 shows the shortest distances between each of the vertices in graph g as returned by `igraph`.

The *average path length* in a graph is determined by calculating the shortest paths between all pairs of vertices. This function does not consider edge weights currently and uses a breadth-first search.

Definition The *eccentricity* of a vertex is calculated by measuring the shortest distance from (or to) the vertex, to (or from) all vertices in the graph, and taking the maximum.

This implementation ignores vertex pairs that are in different components. Isolate vertices have eccentricity zero. Table 5.16 shows the eccentricity of each of the vertices in g. 16 is an isolated vertex and hence has eccentricity zero. The largest distance from vertex 1 is 3 (Row 1 in Table 5.15). Hence eccentricty of vertex 1 is 3. Similarly for other vertices.

Table 5.16: Eccentricity (E) of vertices in g

Vertex	1	2	3	4	5	6	7	8
E	3	3	3	4	4	4	3	4
Vertex	9	10	11	12	13	14	15	16
E	4	5	4	4	5	5	5	0

Definition The *diameter* of a graph is the length of the longest geodesic.

This is also the largest eccentricity value in the graph. `diameter`(g) = 5.

Definition The *radius* of a graph is the smallest eccentricity in it.

`r`adius(g) = 0

Definition Also known as *adhesion*, *edge connectivity* of a pair of vertices (source and target) is the minimum number of edges needed to be removed to eliminate all (directed) paths from source to target.

The edge connectivity of a graph is the minimum of the edge connectivity of every (ordered) pair of vertices in the graph. i.e, it is the minimum number of edges needed to be removed to obtain a graph which is not strongly connected. For graphs that are not connected, edge connectivity is obviously zero.

Definition The *edge density* of a graph also known as *density* of a graph is the ratio of the number of edges and the number of possible edges.

edge_density(g) = 0.1666667

5.4.2 Centrality

Centrality indicates how important a given vertex or an edge is in a graph. Measures of centrality are designed to quantify notions of 'importance'. We first discuss the four main centrality measures, their definitions, and values for the graph (g) followed by other centrality measures.

1. **Vertex centrality**

   The most widely used measure of vertex centrality is the vertex degree of a graph. This is its most basic structural property.

   Definition The degree of a vertex is the number of its adjacent edges.

   Table 5.17: Degree of vertices of g.

Vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Degree	6	3	3	2	1	2	3	4	1	1	3	4	3	1	3	0

   Table 5.17 shows the degree of vertices of g.

   We present the frequency of each of the degrees in Table 5.18.

   The next two centrality measures use the concept of shortest distance and shortest path.

Table 5.18: Frequency of degrees in g

Degree	**Frequency**
0	1
1	4
2	2
3	6
4	2
5	0
6	1

2. **Closeness centrality**

Closeness centrality measures how many steps are required to access every other vertex from a given vertex. It captures the idea that a vertex is 'central' if it is close to many other vertices.

Definition The *closeness centrality* of a vertex is defined by the inverse of the average length of the shortest paths to/from all the other vertices in the graph. It is given by the formula $(1/\sum(d(v,i), i \neq v))$

If there is no (directed) path between vertices v and i, the total number of vertices is used in the formula instead of the path length. Table 5.19 shows the closeness centrality of each of the vertices in g.

Table 5.19: Closeness centrality (C) of vertices in g

Vertex	1	2	3	4	5	6	7	8
C	0.0244	0.0222	0.02222	0.0192	0.0185	0.0208	0.0227	0.0217
Vertex	9	10	11	12	13	14	15	16
C	0.0172	0.0154	0.0192	0.0208	0.0196	0.0169	0.0185	0.0041

We illustrate how the closeness centrality is calculated using vertex 1 as an example. The sum of all shortest paths from vertex 1 to every other vertex in g is 25. (This

is sum of all values in row 1 in Table 5.15). Since there is no path from vertex 1 to vertex 16 (∞), the length of this path is considered to be 16 (the total number of vertices). Thus, the closeness centrality of vertex 1 is the inverse of 41(25 + 16). (1/41 = 0.024390244).

In certain cases where the graph is too large it may not be feasible to compute the shortest path between every pair of vertices. For such scenarios, `igraph` provides a function `estimate_closeness` that only considers paths of up to a certain length (called cutoff). This function can be run for larger graphs (if cutoff is small).

Table 5.20 shows the estimated closeness centrality of each of the vertices, when cutoff is 3. i.e., only paths of up to length 3 are considered.

Table 5.20: Estimated closeness centrality (C_e) of vertices in g with cutoff = 3.

Vertex	1	2	3	4	5	6	7	8
C_e	0.0244	0.0222	0.0222	0.0114	0.0111	0.0167	0.0227	0.0172
Vertex	9	10	11	12	13	14	15	16
C_e	0.0094	0.0068	0.0014	0.0139	0.0016	0.0085	0.0099	0.0042

We consider vertex 4 as an example. The sum of all shortest paths from vertex 4 to every other vertex in g up to length 3, is 24. (This is sum of all values ≤ 3 in row 4 of Table 5.15). All distances > 3 are considered to be 16 (which is equal to total number of vertices). There are 4 such entries in row 4 (total of 64). Thus, the closeness centrality of vertex 4 is the inverse of 24 + 64. (1/80 = 0.0114).

3. **Betweenness centrality**

   Also known as vertex betweenness, betweenness centrality summarizes the extent to which a vertex is located between other pairs of vertices. This is based upon the perspective that 'importance' relates to where a vertex is located with respect to the paths in the network graph. It tells us how much of an intermediary or connector a

vertex is.

Definition *Vertex betweenness* is defined by the number of geodesics (shortest paths) going through a vertex. The vertex betweenness of vertex v is given by $\sum_{i,j} g_{ivj}/g_{ij}$ *where* $i \neq j,\ i \neq v,\ j \neq v$, where g_{ivj} is the number of shortest paths that between i and j that pass through v and g_{ij} is the total number of shortest paths from i to j.

Table 5.21 shows the betweenness centrality of each of the vertices in g.

Table 5.21: Betweenness centrality (B) of vertices in g

Vertex	1	2	3	4	5	6	7	8
B	47.5000	8.3333	19.3333	13.0000	0.0000	8.5000	10.3333	23.1667
Vertex	9	10	11	12	13	14	15	16
B	0.0000	0.0000	5.0000	9.3333	9.5555	0.0000	2.0000	0.0000

We notice from Figure 5.7 that vertices 5, 9, 10, 14, and 16 cannot be in the shortest path between any pair of vertices. Hence their betweenness is 0. We illustrate the computation of betweenness metric with vertex 15 as an example. There are four shortest paths that go through vertex 15. These are shortest paths between -

- Vertices 11 and 13: Shortest paths between 11 and 13 include paths (11, 15, 13) and (11, 12, 13). There are two shortest paths between 11 and 13, one of which passes through 15. Hence $g_{11,15,13} = 1$ and $g_{11,13} = 2$. g_{ivj}/g_{ij} in this case is 1/2 = 0.5.
- Vertices 6 and 13: Shortest paths between 6 and 13 include paths (6, 11, 15, 13) and (6, 11, 12, 13). Again, there are two shortest paths between 6 and 13, one of which passes through 15. Hence $g_{6,11,15,13} = 1$ and $g_{6,13} = 2$. g_{ivj}/g_{ij} is $1/2 = 0.5$.

- Vertices 8 and 11: Shortest paths between 8 and 11 include paths (8, 13, 15, 11) and (8, 13, 12, 11). As in previous cases, $g_{8,13,15,11} = 1$ and $g_{8,11} = 2$. g_{ivj}/g_{ij} is $1/2 = 0.5$.
- Vertices 14 and 11: Shortest paths between 14 and 11 include paths (14, 8, 13, 15, 11) and (14, 8, 13, 12, 11). Again, $g_{14,8,13,15,11} = 1$ and $g_{14,11} = 2$. g_{ivj}/g_{ij} is $1/2 = 0.5$.

Thus, betweenness of vertex 15 is $\sum g_{ivj}/g_{ij} = 2$.

In certain cases where the graph is too large it may not be feasible to compute shortest path between every pair of vertices. For such scenarios, igraph provides a function estimate_betweenness similar to estimate_closeness that only considers paths of up to cutoff. Table 5.22 shows the estimated betweenness centrality of each of the vertices, when cutoff is 2. i.e., only paths of up to length 2 are considered.

Table 5.22: Estimated betweenness centrality (B_e) of vertices in g with cutoff = 2.

Vertex	1	2	3	4	5	6	7	8
B_e	13.5	1.5	2.5	1.0	0.0	1.0	2.0	5.5
Vertex	9	10	11	12	13	14	15	16
B_e	0.0	0.0	2.0	3.5	2.0	0.0	0.5	0.0

Consider example of vertex 8. With shortest path length ≤ 2, the following shortest paths go through vertex 8. These are shortest paths between -

- Vertices 13 and 14: Shortest path between 13 and 14 includes only one path (13, 8, 14). Hence $g_{13,8,14}/g_{13,14} = 1$.
- Vertices 3 and 13: Shortest path between 3 and 13 includes only one path (3, 8, 13). $g_{3,8,13}/g_{3,13} = 1$.
- Vertices 3 and 14: Shortest path between 3 and 14 includes only one path (3, 8, 14). $g_{3,8,14}/g_{3,14} = 1$.

- Vertices 2 and 13: Shortest path between 2 and 13 includes only one path (2, 8, 13). $g_{2,8,13}/g_{2,13} = 1$.
- Vertices 2 and 14: Shortest path between 2 and 14 includes only one path (2, 8, 14). $g_{2,8,14}/g_{2,14} = 1$.
- Vertices 2 and 3: Shortest path between 3 and 13 includes two paths - (2, 8, 3) and (2, 1, 3). Hence $g_{2,8,3}/g_{2,3} = 0.5$.

Thus, estimated betweenness of vertex 8 is $\sum g_{ivj}/g_{ij} = 5.5$.

4. **Eigenvector centrality**

Eigenvector centrality is based on notions of 'prestige' or 'rank'. It seeks to capture the idea that the more central the neighbors of a vertex are, the more central that vertex itself is.

Definition *Eigenvector centrality* scores correspond to the values of the first eigenvector of the graph adjacency matrix; these scores may, in turn, be interpreted as arising from a reciprocal process in which the centrality of each actor is proportional to the sum of the centralities of those actors to which it is connected. i.e., $\lambda C^e(g) = \sum_j g_{ij} C^e_j(g)$ where λ is a proportionality factor. $C^e(g)$ is an eigenvector of g and λ is it's eigen value.

In general, vertices with high eigenvector centralities are those which are connected to many other vertices which are, in turn, connected to many others (and so on).

Table 5.23 shows the eigenvector centrality of each of the vertices in g.

Some other centrality scores that have their basis in eigenvectors and eigenvector centrality are listed below.

- *Bonacich alpha centrality:* The alpha centrality measure is considered as a generalization of eigenvector centerality to directed graphs. Since our graphs are undirected, we do not consider this centrality measure.

Table 5.23: Eigenvector centrality (E_c) of vertices in g

Vertex	1	2	3	4	5	6	7	8
E_c	1.0000	0.7572	0.5669	0.3391	0.3071	0.4920	0.7937	0.6719
Vertex	9	10	11	12	13	14	15	16
E_c	0.1741	0.1041	0.6019	0.8273	0.6572	0.2063	0.6408	0.0000

- *Bonacich power centrality:* Bonacich power measure corresponds to the notion that the power of a vertex is recursively defined by the sum of the power of its neighbors. The nature of the recursion involved is controlled by a power exponent: positive values imply that vertices become more powerful as their neighbors become more powerful (as occurs in cooperative relations), while negative values imply that vertices become more powerful only as their neighbors become weaker (as occurs in competitive or antagonistic relations). The magnitude of the exponent indicates the tendency of the effect to decay across long walks; higher magnitudes imply slower decay. This is more relevant to social networks than to isomer networks.
- *Kleinberg's authority centrality scores:* The authority scores of the vertices are defined as the principal eigenvector of t(A)*A, where A is the adjacency matrix of the graph. This score is mainly for directed graphs and is the same as eigen centrality scores for undirected graphs(matrices).

Observations on centrality: We now look at the top 7 vertices (sorted in descending order) for all centrality metrics as shown in Table 5.24, Table 5.25, Table 5.26, and Table 5.27.

In all four of the centrality measures, we notice that vertex 1 has the highest value. This indicates that vertex 1 is the most central vertex, as is evident from the graph(Figure 5.7). Vertex 1 maps to the Medici family.

Table 5.24: Top 7 vertices of g sorted on vertex centrality

Vertex	Degree
1	6
8	4
12	4
15	3
7	3
3	3
2	3

Table 5.25: Top 7 vertices of g sorted on closeness centrality

Vertex	Clos. cent
1	0.0244
7	0.0227
3	0.0222
2	0.0222
5	0.0217
11	0.0208
6	0.0208

Next, we find the vertices other than 1 that are are at the intersection of the top seven of all four measures These are vertices 8, 12, and 7. These vertices map to the Guadagni, Strozzi, and Ridolfi families that were also prominent in the early fifteenth century. In this example, we see the relevance and importance of centrality measures even before they were recognized as such or calculated.

5.4.3 Centralization

Centralization is a method for creating a graph level centralization measure from the centrality scores of the vertices. We measure how central the most central node in the network is in relation to all other nodes.

Definition *Centralization* can be defined by the formula, $C(g) = \sum_{v}(max(c_w) - c_v)$, where c_v is the centrality of vertex v and $max(c_w)$ is the largest centrality value of all vertices in

Table 5.26: Top 7 vertices sorted of g on betweenness centrality

Vertex	**Bet. cent**
1	47.5000
8	23.1667
3	19.3333
4	13.0000
7	10.3333
13	9.5000
12	9.3333

Table 5.27: Top 7 vertices sorted of g on eigenvector centrality

Vertex	**Eignvector. cent**
1	1.0000
12	0.8273
7	0.7937
2	0.7572
8	0.6719
13	0.6572
15	0.6408

g. (This can be degree, closeness, betweenness, or eigenvector centrality.)

Graph-level centrality score is normalized by dividing by the maximum theoretical score for a graph with the same number of vertices, using the same parameters as the graph. For degree, closeness and betweenness the most centralized structure is some version of the star graph. For eigenvector centrality the most centralized structure is the graph with a single edge (and potentially many isolates).

Normalized graph-level centrality score can be calculated using the following steps.

1. Calculate centrality for every node in graph.

2. Select node that has largest centrality.

3. Calculate the difference between largest centrality and every node.

4. Find the theoretical max for that graph and treat that as the largest centrality. Centrality of every other vertex is considered to b zero. Repeat step 3.

5. Divide the value arrived at in step 3 by the one in step 4 to get the centralization score or graph-level centrality score.

We now consider the example of **Degree centralization**. The degree centralization of graph (g) is 0.2333. It is computed as follows:
$max(degree(\text{g}))$ is 6. (Highest value in Table 5.17). Theoretical max for g which is a star graph with 16 vertices is 240 (16×15).
The numerator is $((6-6)+(6-3)+(6-3)+\cdots+(6-0)) = 56$ and the denominator is 240 (Theoretical max). Degree centralization is 0.2333 (Numerator/Denominator).

Table 5.28 shows centralization measures computed for g.

Table 5.28: Centralization measures computed for g

Centrality measure	**Value**	**Theoretical Max**
Degree	0.2333	240
Closeness	0.1790	7.2414
Betweenness	0.3835	1575
Eigen Vector	0.1790	14

5.4.4 Network cohesion

Network cohesion (the extent to which subsets of vertices are cohesive or connected) is another important aspect of networks. Some metrics we consider under network cohesion are given below.

1. **Transitivity:**

   Transitivity measures the probability that the adjacent vertices of a vertex are connected. This is sometimes also called the clustering coefficient.

   `transitivity`(g) = 0.1914894

2. **Cliques:**

   A clique is a complete subgraph in a graph. Apart from listing the cliques, there are other metrics related to cliques.

   count_max_cliques counts the maximal cliques. (A clique in maximal if it cannot be extended to a larger clique. The largest cliques are always maximal, but a maximal clique is not necessarily the largest.)

   count_maximal_cliques (g) = 16

   We also calculate size of the largest clique(s). clique_num(g) = 3 in our example. We see in graph g that there are three cliques (each of size 3). These are the cliques formed by vertices(1, 2, 7), vertices(11, 12, 15) and vertices(12, 13, 15).

3. **k-core graph:**

   Cliques require that the subgraph is complete. This is often a very rigid condition. This condition is weakened in a k-core graph.

   Definition The $k - core$ of a graph is the maximal subgraph in which every vertex has at least degree k.

   Table 5.29 indicates that graph g contains 3 subgraphs as shown in Figure 5.8. Vertices of each subgraph (where $k = 0$, $k = 1$ and $k = 2$) are shown in different colors.

Table 5.29: k-core vertices of g

Vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Degree	2	2	2	1	1	2	2	2	1	1	2	2	2	1	2	0

4. **Components:**

   components finds the maximal (weakly or strongly) connected components of a graph. The function components returns the number of components, size of each component,

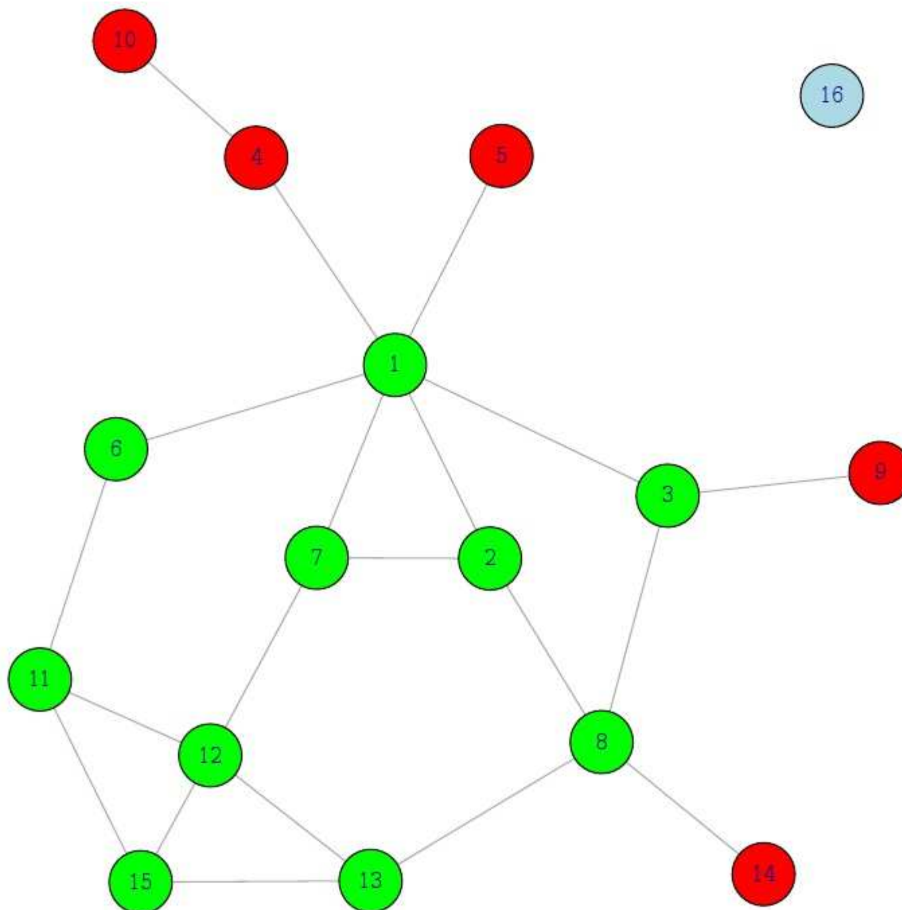


Figure 5.8: Graph showing k-core decomposition of g

and the component each vertex belongs to. In our example, components(g) returns size of components as 15 and 1, number of components as 2, and the component number each vertex belongs to as is shown in Table 5.30.

Table 5.30: Component of the graph (g) each vertex belongs to.

Vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Component	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2

Notice that graph g has two components. The first one consists of vertices 1 though 15, while the second component is only vertex 16. We find the individual components of the graph using the function decompose.graph(). This returns the number of vertices and edges in each component along with the edge list of individual components. The

Table 5.31: Edge list of individual components of g.

[1]	1 –2	1 –3	1 –4	1 –5	1 –6	1 –7	2 –7	2 –8	3 –8
	3 –9	4 –10	6 –11	7 –12	8 –13	8 –14	11–12	11–15	12–13
	12–15	13–15							
[2]									

second row in Table 5.31 is empty since the second component consists of only one vertex (16) and no edges. We can also list the number of components and the number of vertices belonging to each component as follows.

Table 5.32: Number of components and their sizes in g

1	15
1	1

We see in Table 5.32 that one component dominates the other in magnitude. This is a common phenomenon called the *giant component*.

5. **Subcomponents:**

   Subcomponent finds all vertices reachable from a given vertex. This is done by conducting a breadh-first search starting from the given vertex. In graph g, subcomponent of vertex 1 (subcomponent (g, 1)) returns all vertices, while subcomponent of vertex 16 (subcomponent (g, 16)) returns only vertex 16.

6. **Articulation points:**

   Definition Articulation points or cut vertices are vertices whose removal increases the number of connected components in a graph.

   If the original graph is connected, then the removal of a single articulation point makes it unconnected. This is true of components of a graph too. In our example, articulation.points(g) returns vertices 3, 8, 4, and 1. We observe that removing any of these vertices from g, will divide the graph into more connected components.

5.4.5 Community detection

A community is a subset of vertices that are well connected among themselves and at the same time are relatively well separated from the remaining vertices.

Definition [11] Community can be thought of as a graph partitioning problem, where the graph partitioning algorithms seek a partition $C = \{C_1, C_2, \cdots C_k\}$ of the vertex set V of a graph $g = (V, E)$ in such a manner that the sets $E(C_k, C_{k'})$ of edges connecting vertices in C_k to vertics in $C_{k'}$ are relatively small in size compared to the sets $E(C_k) = E(C_k, C_k)$ of edges connecting vertices within the C_k.

Assignment of vertices to communities is *community detection.* The quality of a community is found by comparing the fraction of edges within a cluster to expected fraction if edges were distributed at random. This is also known as *modularity.*

igraph provides the implementation of various community detection algorithms. We now describe a few algorithms and their output for the example graph. Note that communities

and clusters are terms that are used inter-changeably in network analysis. Apart from the igraph manual[6], the following sites have been referred to for an explanation of the algorithms and their run times[34, 35].

1. **Edge betweennness**

   This is a hierarchical decomposition process where edges are removed in the decreasing order of their edge betweenness scores. The idea of the edge betweenness based community structure detection is that it is likely that edges connecting separate modules have high edge betweenness as all the shortest paths from one module to another must traverse through them. So if we gradually remove the edge with the highest edge betweenness score we will get a hierarchical map. The actual algorithm is described in [36]. The runtime of the algorithm is $\mid V \mid\mid E \mid^2$[34].

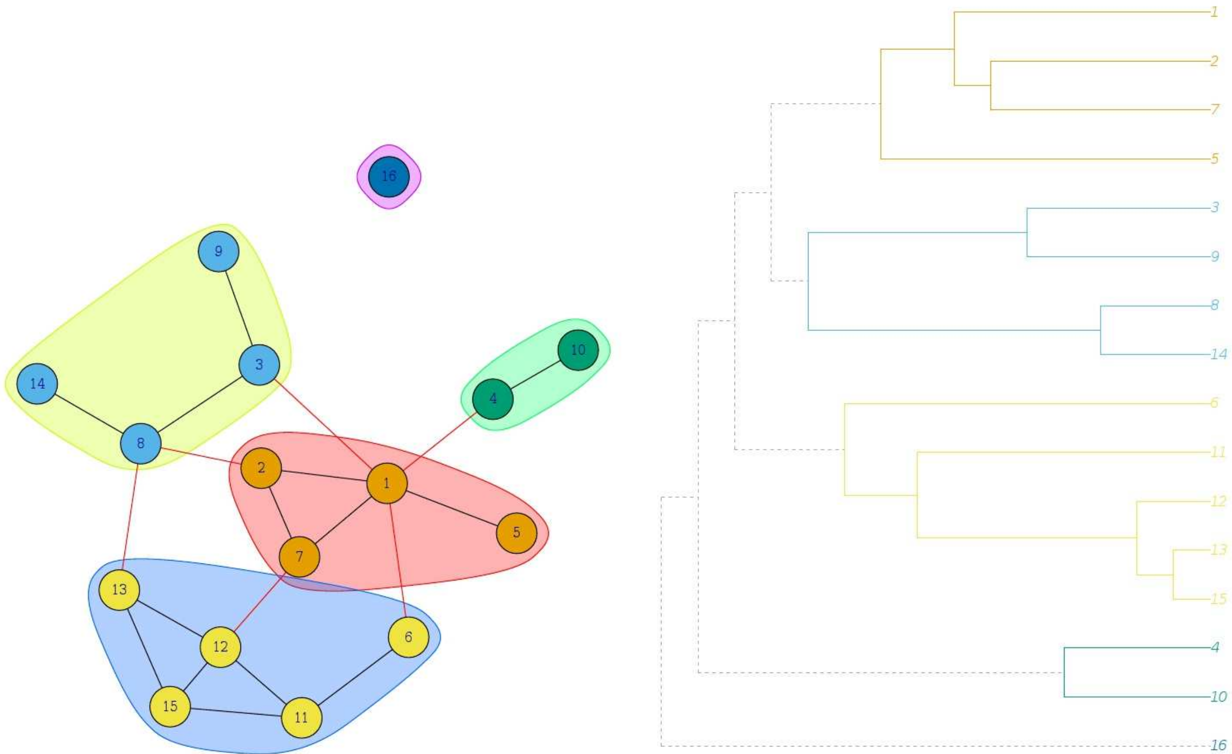


Figure 5.9: Edge betweenness clustering on g

Figure 5.9 shows the communities returned by the edge betweenness clustering algorithm. The dendogram shows the sequence of decomposition of edges. Details of each cluster are shown in Table 5.33. There are 5 clusters. The size of the largest cluster is 5. The modularity of this division is 0.40.

Table 5.33: Communities returned by Edge betweenness clustering algorithm

Cluster Number	1	2	3	4	5
No. of components	4	4	2	5	1

2. **Fast greedy**

   This is a bottom-up hierarchical approach. It tries to optimize modularity in a greedy manner. Initially, every vertex belongs to a separate community, and communities are merged iteratively such that each merge is locally optimal (i.e. yields the largest increase in the current value of modularity). The algorithm stops when it is not possible to increase the modularity any more. This method is fast. The actual algorithm is described in [37]. The runtime of the algorithm is $|V||E|\log|V|$[34].

   Figure 5.10 shows the communities returned by the Fast greedy clustering algorithm. The dendogram shows the sequence of decomposition of edges. Details of each cluster are shown in Table 5.34. There are 5 clusters. The size of the largest cluster is 5. The modularity of this division is 0.40.

Table 5.34: Communities returned by Fast greedy clustering algorithm

Cluster Number	1	2	3	4	5
No. of components	4	4	2	5	1

3. **Walktrap clustering**

   This is also a hierarchical process that tries to find densely connected subgraphs in a

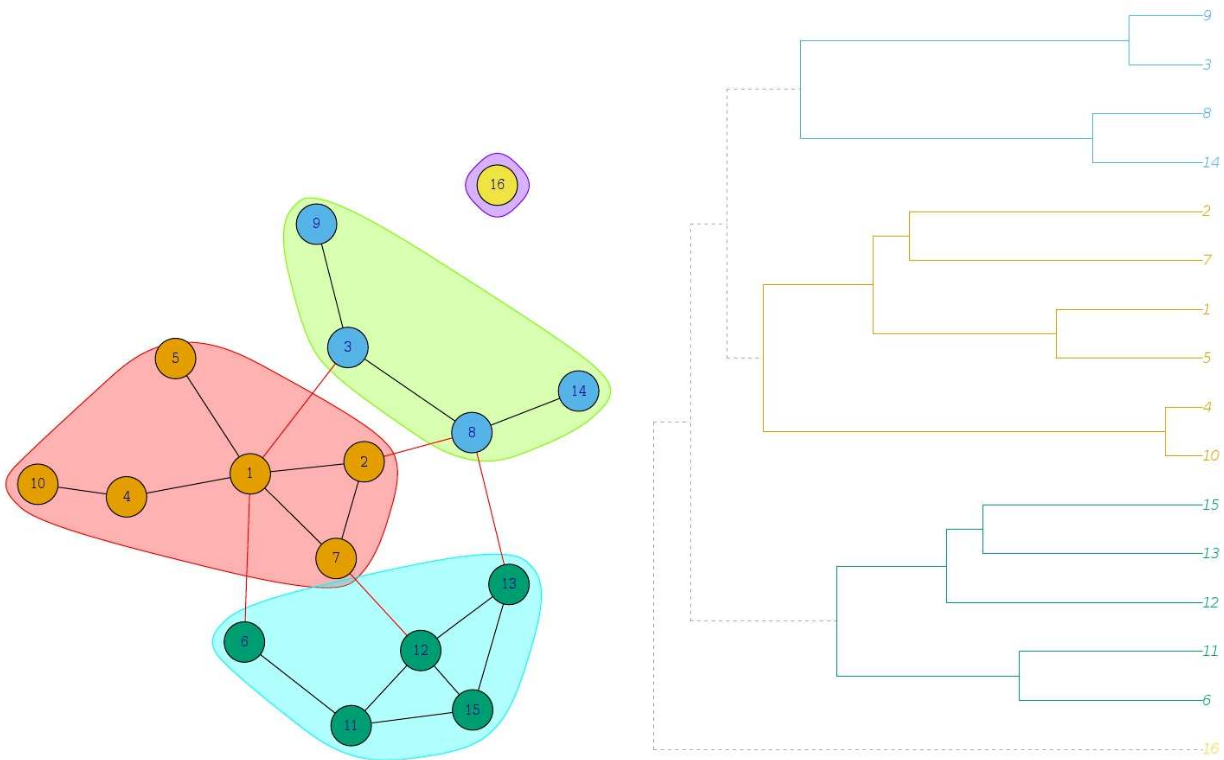


Figure 5.10: Fast greedy clustering on g

graph via random walks. The idea is that short random walks tend to stay in the same community. Walktrap runs short random walks of n steps (default numbr of steps is 4) and uses the results of these random walks to merge separate communities in a bottom-up manner. This is based on the algorithm described in [38]. The runtime of the algorithm is $\mid E \mid\mid V \mid^2$[34].

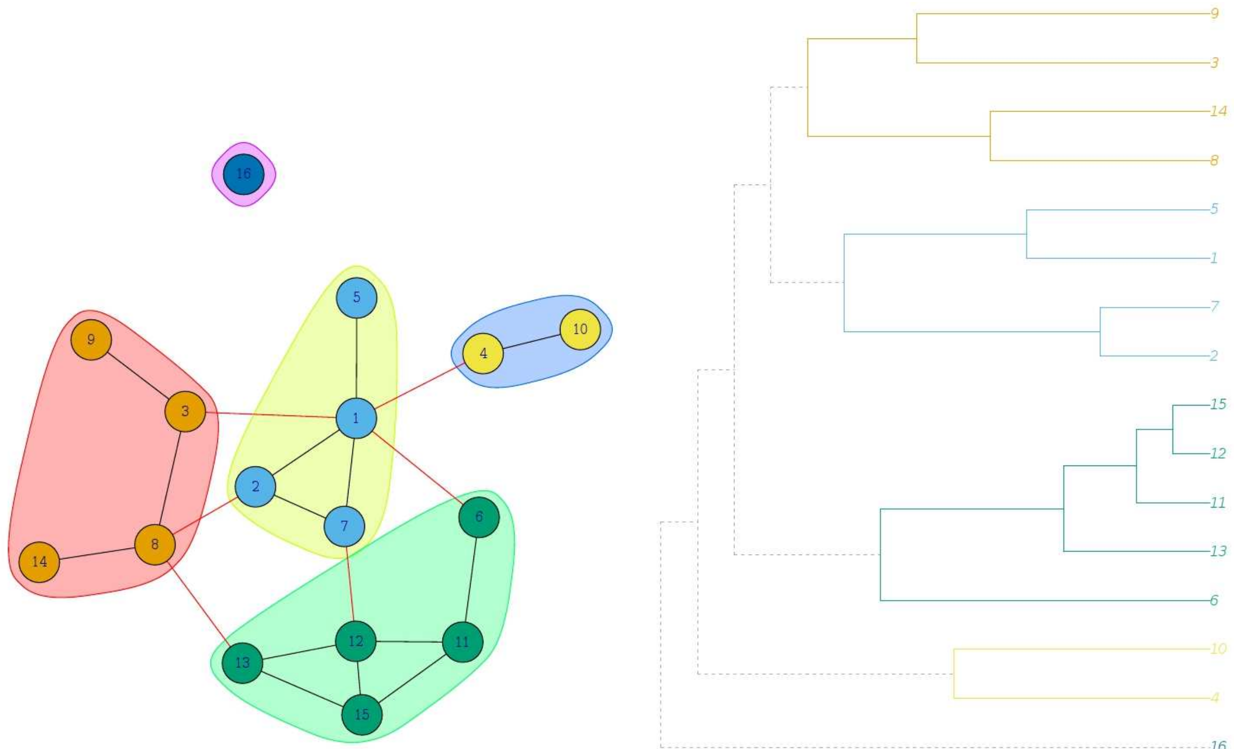


Figure 5.11: Walktrap clustering on g

Figure 5.11 shows the clusters returned by the walktrap community detection algorithm. There are five clusters. Details of each cluster are in Table 5.35. The size of the largest cluster is 5. The modularity of this division is 0.40.

Table 5.35: Communities returned by Walktrap clustering algorithm

Cluster Number	1	2	3	4	5
No. of components	4	4	5	2	1

4. **Community matrix of leading eigenvector based clustering**

   This is also a hierarchical process that tries to find densely connected subgraphs by optimizing modularity. It follows a top down approach. This is based on the algorithm described in [39]. The runtime of the algorithm is $c \mid V \mid^2 + \mid E \mid$[34]. Figure 5.12 shows

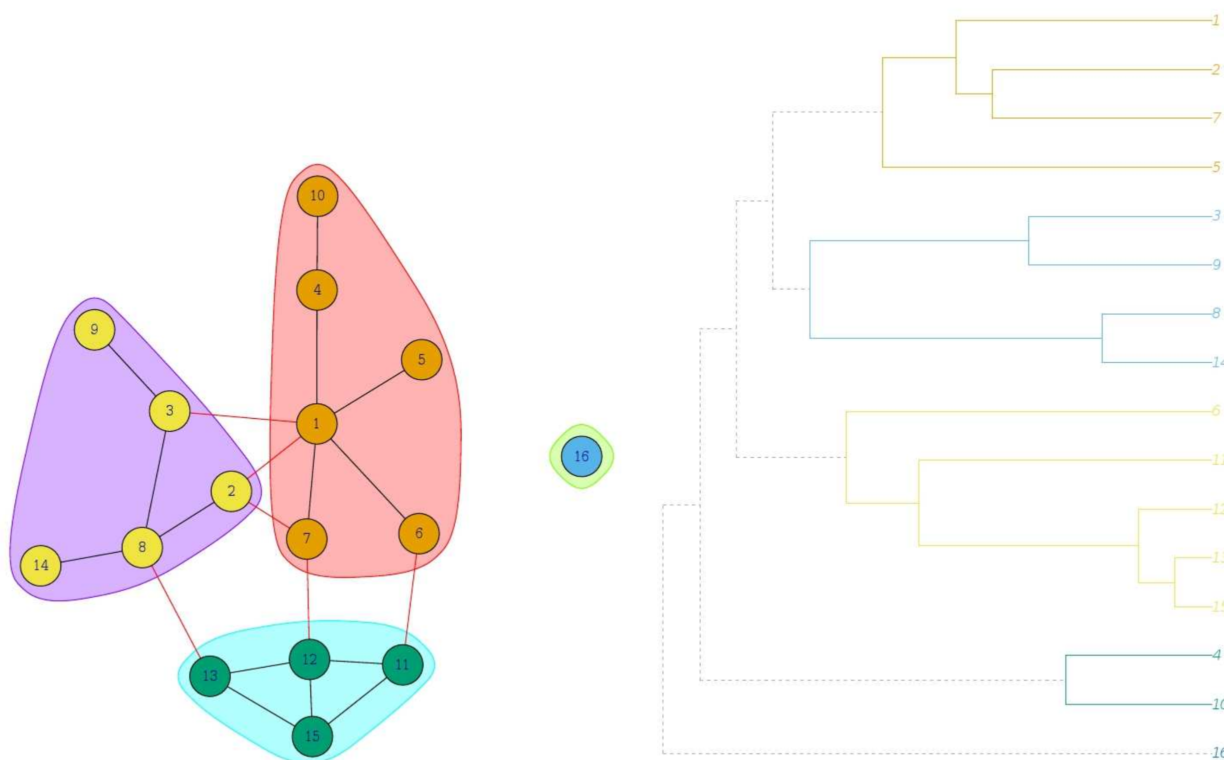


Figure 5.12: Leading eigenvector clustering on g

the clusters returned by the leading eigenvector community detection algorithm. There are four clusters. Details of each cluster are in Table 5.36. The size of the largest cluster is 6. The modularity of this division is 0.36. Leading eigenvector community detection

Table 5.36: Communities returned by Leading eigenvector clustering algorithm

Cluster Number	1	2	3	4
No. of components	6	1	4	5

algorithm does not consider weighted edges. Since the isomer network consists of

weighted edges (number of bonds cut), this is not a suitable algorithm for us.

5. **Label propagation**

   In this algorithm every node is initialized with a unique label and at every step each node adopts the label that most of its neighbors currently have. In this iterative process densely connected groups of nodes form a consensus on a unique label to form communities. The actual algorithm is described in [40]. The runtime of the algorithm is $\mid V \mid + \mid E \mid$[34].

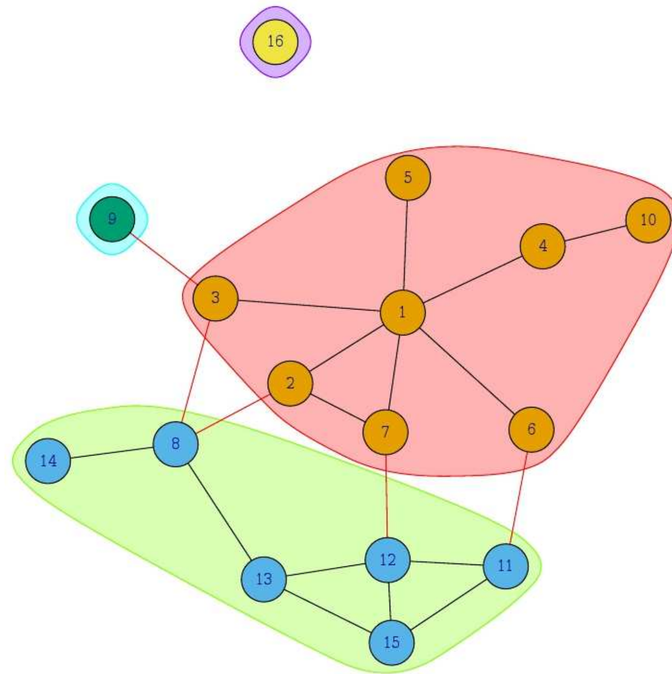


Figure 5.13: Label propagation clustering on g

Figure 5.13 shows the clusters returned by the label propagation community detection algorithm. There are four clusters. Details of each cluster are in Table 5.37. The size of the largest cluster is 8. The modularity of this division is 0.30. A dendogram is not created for this clustering algorithm since it does perform a hierarchical decomposition of the graph.

Table 5.37: Communities returned by Label propagation clustering algorithm

Cluster Number	1	2	3	4
No. of components	8	6	1	1

6. **Infomap** This algorithm recognizes that links in a network induce movement across the network and result in system-wide interdependence. A map equation is used to highlight and simplify the network structure with respect to this flow. The algorithm presents an intuitive derivation of this flow-based and information-theoretic method and provide to efficiently decompose large weighted and directed networks based on the map equationv[41]. The algorithm is given in [41].

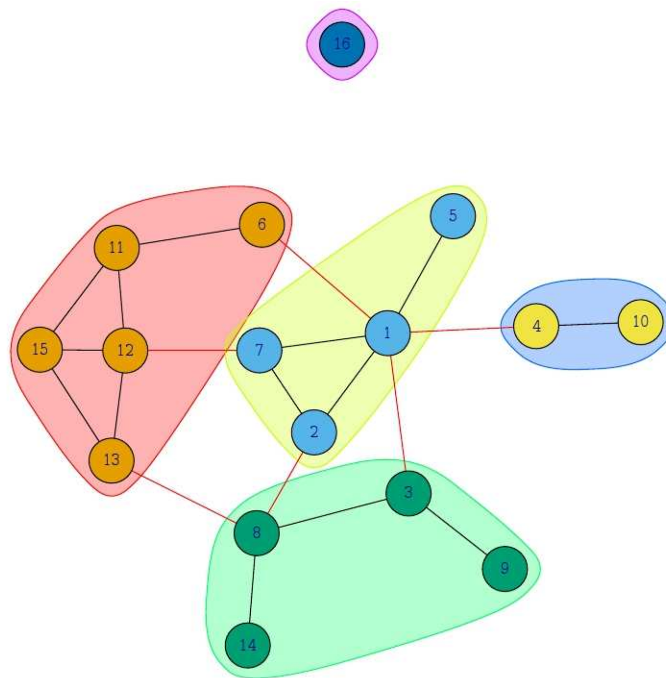


Figure 5.14: Infomap clustering on g

Figure 5.14 shows the clusters returned by the Infomap community detection algorithm. There are four clusters. Details of each cluster are in Table 5.38. The size of the largest cluster is 5. The modularity of this division is 0.40.

Table 5.38: Communities returned by Infomap clustering algorithm

Cluster Number	1	2	3	4	5
No. of components	5	4	4	2	1

Though this algorithm can be run on undirected graphs, it works better on directed graphs.

7. **Louvain clustering (Multi level community clustering)**

This community detection algorithm is based on the modularity measure and a hierarchal approach. Initially, each vertex is assigned to a community on its own. In every step, vertices are re-assigned to communities in a local, greedy way: each vertex is moved to the community with which it achieves the highest contribution to modularity. When no vertices can be reassigned, each community is considered a vertex on its own, and the process starts again with the merged communities. The process stops when there is only a single vertex left or when the modularity cannot be increased any more in a step. The algorithm is described in [42]. The runtime of the algorithm is "linear" when $\mid V \mid \approx \mid E \mid$[34].

Figure 5.15 shows the clusters returned by the Louvian community detection algorithm. There are five clusters. Details of each cluster are in Table 5.39. The size of the largest cluster is 5. The modularity of this division is 0.87. Though this cluster uses a hierarchal approach, no dendogram is created.

Table 5.39: Communities returned by Louvain clustering algorithm

Cluster Number	1	2	3	4	5
No. of components	5	2	4	4	1

8. **Optimal clustering**

This algorithm calculates the optimal community structure for a graph, in terms of

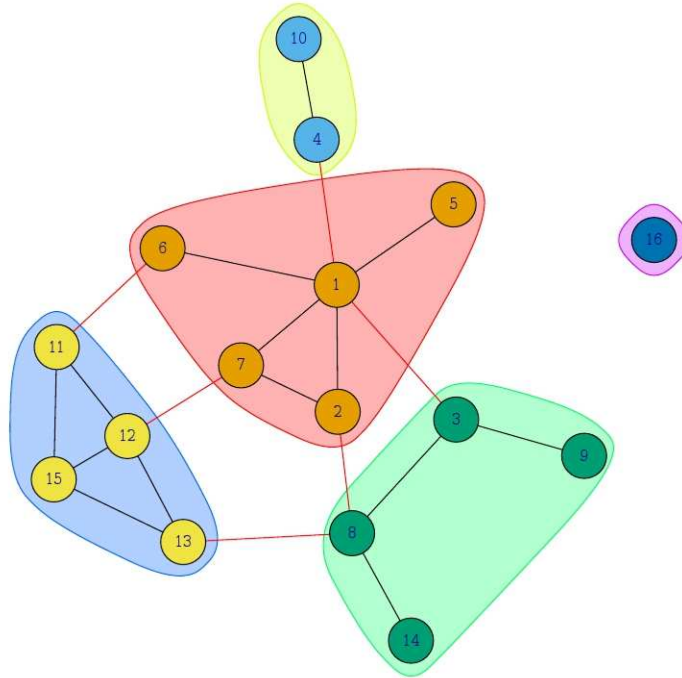


Figure 5.15: Louvain clustering on g

maximal modularity score. The calculation is done by transforming the modularity maximization into an integer programming problem, and then calling the GLPK library to solve that. The algorithm is given in [43].

Figure 5.16 shows the clusters returned by the Optimal community detection algorithm. There are four clusters. Details of each cluster are in Table 5.40. The size of the largest cluster is 6. The modularity of this division is 0.40.

Table 5.40: Communities returned by Optimal clustering algorithm

Cluster Number	1	2	3	4
No. of components	6	4	5	1

Since modularity optimization is an NP-complete problem, and all known algorithms for it have exponential time complexity, this clustering algorithm is not suitable for large graphs.

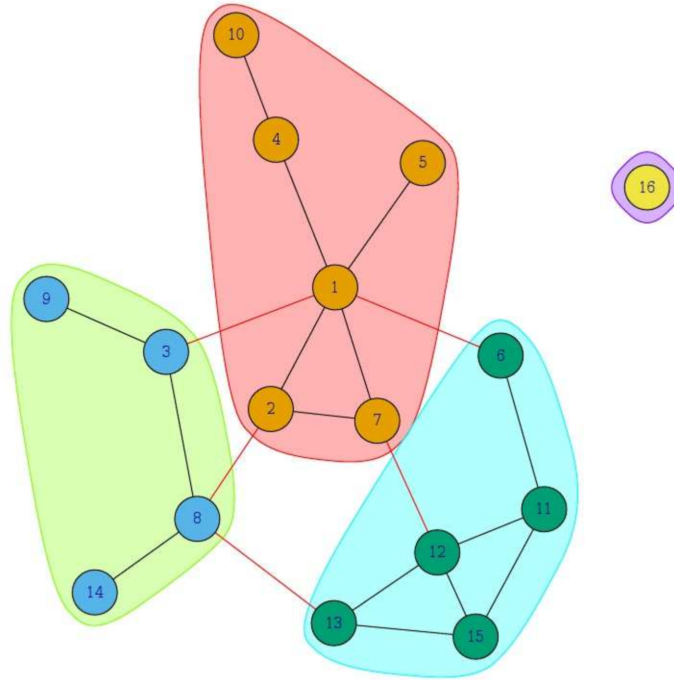


Figure 5.16: Optimal clustering on g

Table 5.41 lists the number of clusters, size of largest cluster, and modularity for each of the algorithms.

Table 5.41: Result of community detection algorithms run on g

Algorithm	**Number of clusters**	**Size of largest cluster**	**Modularity**
Edge betweenness	5	5	0.40
Fast greedy	4	6	0.40
Walktrap clustering	5	5	0.40
Leading eigenvector	4	6	0.36
Label propagation	4	8	0.30
Infomap	5	5	0.40
Louvian	5	5	0.87
Optimal	4	6	0.40
Spinglass	Does not work for unconnected graphs		

CHAPTER 6

CONCLUSION

In this chapter we present a summary of the contributions of this dissertation and also the future directions in this research.

6.1 Summary

To summarize, the contributions of this dissertation are a variety of techniques that can be employed to more efficiently (with respect to time and memory) compute isomer networks.

Specifically, our contributions are as listed below.

1. We improved the network extraction process (All Pairs SAA) by using the symmetry present in most molecules to reduce runtime and memory and streamlining the algorithm used for the detection of duplicate dnNames[1], a key step in determining the bond count distances between pairs of isomers. Together, these techniques resulted in reductions in memory of up to 60% and improvements in runtime of up to a factor of 100.

2. We developed an algorithm that enables fragments of the isomer network to be developed in groups that can be accommodated within the available memory resources, such that every isomer is in a group with every other isomer once and only once. The algorithm provides a solution to sub divide the “big data” problem that arises in the construction of isomer networks into several independent “small data” problems. This grouping technique helped divide large data sets into independent smaller ones that could be processed in parallel.

3. We generated the isomer network for 1,050,125 isomers of Nicotine using the cloud computing capabilities of Amazon Web Services[2] and Microsoft Azure[3]. We presented some graph analytical metrics that can be used to analyze such a network and

our results from analyzing a network of approximately 500,000 and 1,000,000 isomers using the metrics described.

6.2 Future directions

We now present future directions this research can take.

1. Work with the chemists at Chemical Space Project[4] to understand significance of results in the cheminformatics space, and provide further insights into the network based on their feedback and requirements.

2. Explore the use of other graph analytical tools that can be used to facilitate Step 1.

3. Generate isomer networks for other compounds (like Phenmetrazine and Tyrosine).

4. In [44, 45], the authors develop visualization tools for automated reaction mapping. We would like to develop a similar tool for displaying the results of the isomer analysis.

REFERENCES CITED

[1] Tina M. Kouri, Mahendra Awale, James K. Slyby, Jean-Louis Reymond, and Dinesh P. Mehta. "social" network of isomers based on bond count distance: Algorithms. *J. Chem. Inf. Model.*, 54(1):57–68, 2014. doi: 10.1021/ci4005173. URL http://dx.doi.org/10.1021/ci4005173. PMID: 24350890.

[2] Amazon Web Services. AWS. URL https://aws.amazon.com/.

[3] Microsoft Azure. https://azure.microsoft.com/en-us/.

[4] Jean-Louis Reymond. The chemical space project. *Acc. Chem. Res.*, 48(3):722–730, 2015.

[5] J.-L. Reymond, R. van Deursen, L.C. Blum, and L. Ruddigkeit. Chemical space as a source for new drugs. *Med. Chem. Commun.*, 1:30–38, 2010. doi: 10.1039/C0MD00020E. URL http://dx.doi.org/10.1039/C0MD00020E.

[6] Mark S. Handcock, David R. Hunter, Carter T. Butts, Steven M. Goodreau, and Martina Morris. *statnet: Software tools for the Statistical Modeling of Network Data*. Seattle, WA, 2003. URL http://statnetproject.org.

[7] *Network Analysis: Methodological Foundations*. Springer Berlin Heidelberg.

[8] Matthew O Jackson. *Social and Economic Networks*. Princeton University Press, 2010.

[9] Mark Newman. *Networks: An Introduction*. OUP Oxford, 2010.

[10] Nagiza F Samatova, William Hendrix, John Jenkins, Kanchana Padmanabhan, and Arpan Chakraborty. *Practical Graph Mining with R*. CRC Press, 2013.

[11] E.D. Kolaczyk and G. Csárdi. *Statistical Analysis of Network Data with R*. Use R! Springer New York, 2014. ISBN 9781493909827. URL https://books.google.com/books?id=ZaR8oAEACAAJ.

[12] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism ii. *Journal of Symbolic Computation*, 60:94–112, 2014. ISSN 0747-7171. doi: http://dx.doi.org/10.1016/j.jsc.2013.09.003. URL http://www.sciencedirect.com/science/article/pii/S0747717113001193.

[13] B. McKay. Practical Graph Isomorphism. *Congr. Numer.*, 30:45–87, 1981.

[14] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Graph symmetry detection and canonical labeling: Differences and synergies. In Andrei Voronkov, editor, *Turing-100*, volume 10 of *EPiC Series*, pages 181–195. EasyChair, 2012.

[15] Tommi Junttila and Petteri Kaski. *Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs*, chapter 12, pages 135–149. . doi: 10.1137/1.9781611972870.13. URL http://epubs.siam.org/doi/abs/10.1137/1.9781611972870.13.

[16] Tommi Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In Alberto Marchetti-Spaccamela and Michael Segal, editors, *Theory and Practice of Algorithms in (Computer) Systems*, volume 6595 of *Lecture Notes in Computer Science*, pages 151–162. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19753-6. URL http://dx.doi.org/10.1007/978-3-642-19754-316.

[17] Tommi Junttila and Petteri Kaski. bliss: A tool for computing automorphism groups and canonical labelings of graphs, . URL http://www.tcs.hut.fi/Software/bliss.

[18] Paul T Darga, Mark H Liffiton, Karem A Sakallah, and Igor L Markov. Exploiting structure in symmetry detection for cnf. In *Proceedings of the 41st annual Design Automation Conference*, pages 530–534. ACM, 2004.

[19] PT Darga, KA Sakallah, and IL Markov. Faster symmetry discovery using sparsity of symmetries. In *2008 45th ACM/IEEE Design Automation Conference*.

[20] Fernández Anta Antonio López-Presa, José Luis. Fast algorithm for graph isomorphism testing. In Jan Vahrenhold, editor, *Experimental Algorithms*, volume 5526 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02010-0. doi: 10.1007/978-3-642-02011-721. URL http://dx.doi.org/10.1007/978-3-642-02011-721.

[21] José Luis López-Presa, Antonio Fernández Anta, and Luis Núñez Chiroque. Conauto-2.0: Fast isomorphism testing and automorphism group computation. *CoRR*, abs/1108.1060, 2011. URL http://arxiv.org/abs/1108.1060.

[22] Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR*, abs/0804.4881, 2008. URL http://arxiv.org/abs/0804.4881.

[23] Daniel Eli (http://math.stackexchange.com/users/9899/danieleli).

[24] Jonathan L. Gross. *Combinatorial Methods with Computer Applications*. Chapman and Hall/CRC 2007. doi: 10.1201/b15899-12.

[25] Kenneth H Rosen. *Handbook of discrete and combinatorial mathematics*. CRC press, 1999.

[26] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994. ISBN 0201558025.

[27] Amazon Web Services EC2 documentation. `https://aws.amazon.com/ec2/getting-started/`.

[28] Amazon Web Services S3 documentation. `https://aws.amazon.com/documentation/s3/`.

[29] FileZilla. `https://filezilla-project.org/`.

[30] Microsoft Azure documentation. `https://docs.microsoft.com/en-us/azure/`.

[31] Microsoft Azure Linux Virtual Machines documentation. `https://docs.microsoft.com/en-us/azure/virtual-machines/linux/`.

[32] M.O. Jackson. *Social and Economic Networks*. Princeton University Press, 2010. ISBN 9781400833993. URL `https://books.google.com/books?id=rFzHinVAq7gC`.

[33] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0945-3245. doi: 10.1007/BF01386390. URL `http://dx.doi.org/10.1007/BF01386390`.

[34] Community detection algorithm complexity and specifications. `https://www.r-bloggers.com/summary-of-community-detection-algorithms-in-igraph-0-6/`.

[35] Description of community detection algorithms. http://stackoverflow.com/questions/9471906/what-are-the-differences-between-community-detection-algorithms-in-igraph.

[36] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69:026113, Feb 2004. doi: 10.1103/PhysRevE.69.026113. URL `https://link.aps.org/doi/10.1103/PhysRevE.69.026113`.

[37] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004. doi: 10.1103/PhysRevE.70.066111. URL `https://link.aps.org/doi/10.1103/PhysRevE.70.066111`.

[38] Pascal Pons and Matthieu Latapy. *Computing Communities in Large Networks Using Random Walks*, pages 284–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32085-2. doi: 10.1007/11569596_31. URL `http://dx.doi.org/10.1007/11569596_31`.

[39] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Phys. Rev. E*, 74:036104, Sep 2006. doi: 10.1103/PhysRevE.74.036104. URL `https://link.aps.org/doi/10.1103/PhysRevE.74.036104`.

[40] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007. doi: 10.1103/PhysRevE.76.036106. URL `https://link.aps.org/doi/10.1103/PhysRevE.76.036106`.

[41] M. Rosvall, D. Axelsson, and C. T. Bergstrom. The map equation. *The European Physical Journal Special Topics*, 178(1):13–23, 2009. ISSN 1951-6401. doi: 10.1140/epjst/e2010-01179-1. URL `http://dx.doi.org/10.1140/epjst/e2010-01179-1`.

[42] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008. URL `http://stacks.iop.org/1742-5468/2008/i=10/a=P10008`.

[43] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Trans. on Knowl. and Data Eng.*, 20(2):172–188, February 2008. ISSN 1041-4347. doi: 10.1109/TKDE.2007.190689. URL `http://dx.doi.org/10.1109/TKDE.2007.190689`.

[44] J. Crabtree and D. Mehta. Automated reaction mapping. *J. Exp. Algorithmics*, 13:15:1.15–15:1.29, February 2009.

[45] J. Crabtree, D. Mehta, and T. Kouri. An Open-Source Java Platform for Automated Reaction Mapping. *J. Chem. Inf. Model.*, 50(9):1751–1756, 17 August 2010.