Set Labelling Vertices and Study of Auxiliary Graphs

by

MAHIPAL PRITHVISINH JADEJA
201221015

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

DOCTOR OF PHILOSOPHY

to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

February, 2018

Declaration

I hereby declare that

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

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

Mahipal Prithvisinh Jadeja

Certificate

This is to certify that the thesis work entitled Set Labelling Vertices and Study of Auxiliary Graphs has been carried out by MAHIPAL PRITHVISINH JADEJA (201221015) for the degree of Doctor of Philosophy at *Dhirubhai Ambani Institute of Information and Communication Technology* under our supervision.

Prof. Rahul Muthu
Thesis Supervisor

Prof. Srikrishnan Divakaran
Thesis Co-Supervisor

Acknowledgments

I would like to thank my thesis supervisor Prof. Rahul Muthu, without whom I wouldn't have been able to deliver quality research work. He believed in my abilities throughout my Ph.D. journey. He was always available for discussions. I can't remember even a single instance where I wanted to meet him and couldn't due to his unavailability! Even during weekends and holidays, we used to meet and discuss. He is selfless and always wants his students to be successful even more than himself. I still remember the long rigorous discussions with him during my initial Ph.D. days when I was struggling to get new results. From the day of my interview at DA-IICT till today, he has been a great source of motivation and inspiration for me.

I extend my gratitude to my co-supervisor Prof. Srikrishnan Divakaran (who was also my mentor during my Ph.D. course work) for his valuable suggestions during the research progress seminars and presentations of self-study course subjects. He helped me a lot in choosing my problem statement by discussing about various open problems and their difficulty levels. I was also fortunate enough to work with him as a T.A. for 3 semesters and it was a good learning experience for me which helped me a lot in improving my teaching skills.

I would like to thank my informal supervisor Prof. V. Sunitha for her insightful suggestions during my entire Ph.D. journey. She used to share literature related to my problem. She helped me a lot in improving my writing skills. I am grateful to her for suggesting me appropriate conferences for submitting research papers. I enjoyed working as a Tutor with her for 3 semesters where I also got an opportunity to attend her lectures.

I am grateful to Prof. Jaideep Mulherkar for his valuable suggestions during

research progress seminars. He has encouraged me a lot to do quality research throughout my Ph.D. journey. I was able to learn about group theory and linear algebra in detail while working with him as a Tutor.

I would like to thank other instructors including Prof. Anish Mathuria, Prof. Asim Banerjee and Prof. Suman Mitra for giving me an opportunity to work with them as a T.A. I am also very grateful to Prof. Anish Mathuria and Prof. Punit Bhateja for their insightful suggestions during my synopsis. I would like to thank Prof. Nutan Limaye (Associate Professor, IIT Bombay) and Prof. Yannis Manoussakis (Director of LRI, University Paris-Sud) for appreciating my thesis work and providing valuable inputs. I am thankful to DA-IICT for providing me the opportunity to pursue Ph.D.

I would like to thank former M.Tech. students at DA-IICT-Ankit, Brijesh, Dhaval, Hiral, Rutvi, Falak and Krunal for many fruitful discussions. I would like to thank Kesha Shah (Morgan Stanley, Bengaluru) for informing me about the SIGIR conference. Special thanks to Hitarth Kanakia (Media.net, Mumbai) for writing code of more than 1k lines for the implementation of my proposed visualization technique. I would like to thank my friends including Vishv, Ujash, Dr. Milind, Jay and Jecky for their support. I would also like to thank my hostel roommates in this journey Dr. Ashish Phophalia, Hardik Sailor, Hardik Gajera and Nachiket Desai. On a broader note, I would like to thank all the people who have helped me directly or indirectly in my Ph.D. research work.

I warmly thank my brother Dr. Yashpal and my sister-in-law Bindu. Finally, I would like to thank my father Dr. Prithvisinh and my mother Jyoti for their constant support and motivation. They have always given me strength to handle difficult situations. They have made sure that I do the ’right’ thing in each situation. Their positive attitude and belief in me and my abilities made my Ph.D. journey smooth and stress-less. Lastly, I would like to thank the god, the supreme power, for giving me abilities to do research and providing me this opportunity to use them in the right manner.

Contents

Abstract x

List of Tables xi

List of Figures xii

1 Introduction 1

1.1 Definitions and notations . . . 2

1.1.1 Graph classes . . . 3

1.2 Statements of problem variants . . . 5

1.2.1 Objectives . . . 6

1.3 Set theoretic definitions of total graphs and spanning tree auxiliary graphs . . . 7

1.3.1 Total graphs and set labelliing . . . 7

1.3.2 Spanning tree auxiliary graph and set labelling . . . 8

1.4 Related work . . . 9

1.4.1 Set labelling and its applications . . . 9

1.4.2 Auxiliary graphs . . . 11

1.5 Thesis outline . . . 12

2 Results on Universe Size Number (USN) 14

2.1 Introduction . . . 14

2.1.1 General results on USN . . . 15

2.2 Results on USN for some special families of graphs . . . 16

2.2.1 Grid graph and ladder graph . . . 25

2.3 Cartesian product based method . . . 29

2.4 Results on USN for complete k-ary trees 30
2.4.1 Discussion on additional features 36
2.5 Conclusions 37

3 Labeled Object Treemap: A New Technique For Visualizing Multiple Hierarchies 39
3.1 Introduction 40
3.1.1 Treemap 41
3.2 Tree-map: a visualization tool for large data 44
3.2.1 Guidelines for tree map design 44
3.2.2 Expressive power of tree-map 45
3.2.3 Social network data and tree-map 46
3.2.4 Types of queries supported by tree-maps 48
3.2.5 Tree-map for Twitter data visualization 48
3.2.6 Discussion on interactivity of tree-maps 50
3.3 Visualizing multiple hierarchies 51
3.3.1 Related work 53
3.4 Description of our proposed method and its advantages 53
3.5 Comparison with existing methods 55
3.6 Interactive labeled object treemap 57
3.6.1 Key features of our implementation 59
3.7 Conclusions and future work 62

4 Edgeless Graph: A New Graph Based Information Visualization Technique 64
4.1 Introduction 64
4.2 Effects of dynamic changes (adding and/or removal of vertices and/or edges) on USN 65
4.3 Our proposed methods for graph visualization 70
4.4 Application: Social network analysis 71
4.4.1 Study of dynamic changes 75
4.5 Conclusions and future Work 76

5 Results on UUSN, ILN and UILN **78**

5.1 Introduction . . . 78

5.1.1 General results on UUSN and ILN . . . 79

5.2 Results on UUSN and ILN for some special families of graphs . . . 84

5.2.1 Cartesian product based method . . . 94

5.3 Conclusions and future Work . . . 95

6 Total Graphs **97**

6.1 Introduction . . . 97

6.1.1 Related work . . . 98

6.1.2 Total graphs and set labelliing . . . 98

6.2 Definitions and notation . . . 99

6.3 Total graphs: Basic properties . . . 101

6.3.1 Vertex degrees of $H = T(G)$ in terms of vertex degrees of G. 103

6.3.2 Relationship between the number of vertices and edges of the original graph and its total graph . . . 105

6.4 Results on complete graphs . . . 106

6.4.1 Direct method for construction of total graph of a complete graph ($T(K_n)$) . . . 108

6.5 Characterisation of total graphs and computing the inverse total graph . . . 110

6.6 Conclusions & future directions . . . 114

7 Spanning Tree Auxiliary Graph **115**

7.1 Introduction . . . 116

7.2 Definitions . . . 117

7.3 Parameters . . . 121

7.4 Classification of maximal cliques in $Aux(G)$ in terms of structures in G . . . 122

7.5 Minimal preimage and multiple preimages . . . 124

7.6 Characterisation of Aux . . . 129

7.6.1 Algorithm for computing a basic preimage . . . 129

7.6.2 Analysis 131
7.7 Conclusions 131

8 Conclusions **133**

Bibliography **159**

Abstract

Given a set of nonempty subsets of some universal set, their **intersection** graph is defined as the graph with one vertex for each set and two vertices are adjacent precisely when their representing sets have non-empty intersection. Sometimes these sets are finite, but in many well known examples like geometric graphs (including interval graphs) they are infinite. One can also study the reverse problem of expressing the vertices of a given graph as distinct sets in such a way that adjacency coincides with intersection of the corresponding sets. The sets are usually required to conform to some template, depending on the problem, to be either a finite set, or some geometric set like intervals, circles, discs, cubes etc. The problem of representing a graph as an intersection graph of sets was first introduced by Erdos et al. and they looked at minimising the underlying universal set necessary to represent any given graph. In that paper it was shown that the problem is NP complete. In this thesis, we study a natural variant of this problem which is to consider graphs where vertices represent **distinct** sets and adjacency coincides with **disjointness**. Although this is **nearly** the same problem on the complement graph, for specific families of graphs this is a more natural way of viewing it. The parameters we take into account are the minimum universe size possible (USN), the minimum individual label size possible (ILN) and their uniform versions (UUSN and UILN respectively).

We propose two applications related to information visualization which use the same underlying idea: assignment of unique labels to each vertex of a graph (or tree) and removal of all edges. A pair of nodes is adjacent if and only if their corresponding labels are disjoint. The proposed labelling scheme can be used to establish isomorphism and study of ontologies.

Information visualization is a class of techniques which is used to present data in a graphical or pictorial format. Identification of new patterns as well as an understanding of difficult concepts is possible with the proper use of visualization. Using interactive visualization, various other details related to the information can be obtained. We consider trees in which each node is related to a leaf node (object) of taxonomy. We propose a new technique of visualization namely **Labeled Object Treemap** for the visualization of multiple hierarchies. The comparison of our proposed technique with already known techniques is also made. The total number of distinct elements used in the underlying labelling is asymptotically minimised in the case of k-ary object trees.

The motivation for selecting set labelling is to use cardinalities of labels to identify level numbers of the underlying binary tree using which it will be easier to discover adjacency as well as non-adjacency for all vertices. Our main contribution is the development of a new visualization technique which solves the issues of edge crossing and continuity of the 'Trees in a treemap' visualization technique, while maintaining all the good characteristics of existing methods for visualizing multiple hierarchies. We have also implemented an interactive version of the proposed visualization technique with various features and studied various aspects of treemapping.

Graph Visualization is one of the sub-fields of information visualization. It is used for the visualization of structured data. i.e. for inherently related data elements. In the traditional graph visualization techniques, nodes are used to represent data elements whereas edges are used to represent relations. According to us, the key challenges for any graph based visualization technique are related to the edges. Some of the challenges are planar representation, minimisation of edge crossing, minimising the number of bends, distinguish between the vertices and the edges. The biggest advantage of our proposed representations is that they don't have edges so we don't need to consider the challenges related to edges. An algorithm for obtaining a valid labelling as well as procedures related to dynamic changes (addition/removal of edges and/or vertices) are explained in detail. Space complexities of the proposed methods are $O(n^2)$ and $O(n^3)$ where n

denotes the number of nodes. Application of our proposed methods in the analysis of a social network site is also demonstrated. Characteristics of these methods are highlighted along with future possible modifications.

Graphs constructed to translate some graph problem into another graph problem are usually called auxiliary graphs. Specifically, total graphs of simple graphs are used to translate the total coloring problem of the original graph into a vertex coloring problem of the transformed graph. We obtain a new characterisation of **total graphs** of simple graphs. We also design algorithms to compute the inverse total graph when the input graph is a total graph. These results improve upon the work of Behzad (in terms of simplicity of the algorithm, not running time), by using novel observations on the properties of the local structure in the neighbourhood of each vertex. We obtain direct constructive results for total graph of complete graphs.

The second class of auxiliary graphs which we consider is based on the set of spanning trees of the given graph and the edges constituting those spanning trees. We call this class **spanning tree auxiliary graphs** of simple graphs. Since the class of spanning tree auxiliary graphs of graphs do not have unique preimages (the forward function is not injective), we derive precisely the classes of graphs which have the same auxiliary graph. We design algorithmic ideas for computing a basic preimage and define rules to get other solutions for the same auxiliary graph. We also obtain several results expressing parameters of the auxiliary graph in terms of (not necessarily the same) parameters of the original graph.

Keywords:

Set labelling; intersection number; intersection graphs; vertex labelling; knesser graphs; information visualization; treemapping; graph drawing; taxonomy; visualizing multiple hierarchies; graph visualization; graph labelling; social network analysis; total graph; line graph; auxiliary graph; spanning trees; spanning tree auxiliary graph; blocks; 2-connected graphs; cartesian product.

List of Tables

3.1 Comparison with existing techniques 57

8.1 Summary of results 133

List of Figures

1.1 Input graph G 7

1.2 Total graph of G $(T(G))$ 8

1.3 Construction of $Aux(G)$ from G 8

2.1 Input graph H and $\text{USN}(H) = 2$ 15

2.2 USN $(M_6) = 4$ 17

2.3 USN $(\overline{K_8}) = 4$ 18

2.4 Summary of add-edge procedure 18

2.5 Add-edge procedure : P_7 to P_{11} 20

2.6 Hasse diagram type labelling of Q_4 using the set {5,6,7,8} 22

2.7 Hasse diagram type labelling of Q_4 using 3 disjoint sets with the same cardinality (4) 22

2.8 Step 1 23

2.9 Step $2(a)$ 24

2.10 Final labelling after Step 2b 25

2.11 Structrual similarity of $V_{i,k}$ and $V_{i+1,k-1}$ 26

2.12 Justification of 3rd new element 26

2.13 Base case 27

2.14 Step 1 27

2.15 Step 2 27

2.16 Final labelling 28

2.17 $G_{3,9}$ after step 1 28

2.18 Final labelling $G_{3,9}$ 28

2.19 $G_{4,9}$ after steps 1 and 2 29

2.20 Final labelling $G_{4,9}$ 29
2.21 Cartesian Product Based Method 30
2.22 Summary of adjacency and non-adjacency for L_{N+1} and L_{N-1} . . . 31
2.23 Input 31
2.24 After Step 1 and 2 32
2.25 After Step 3 33
2.26 Output 34
2.27 After 2nd iteration 35
2.28 Input for the modified algorithm 37

3.1 Taxonomy 40
3.2 Hierarchical data and corresponding tree representation 41
3.3 Venn diagram and nested tree-Map 42
3.4 Tree-Map 42
3.5 www.peets.com 43
3.6 www.newsmap.jp 46
3.7 Social CRM tree-Map 47
3.8 Basic Twitter treemap 49
3.9 Twitter treemap with additional information (actor's interests) . . 50
3.10 Twitter treemap integrated with network diagram 50
3.11 a) Two separate tree diagrams and b) Linked tree diagram 52
3.12 Trees in a treemap visualization technique (from [12]) 54
3.13 Our proposed method: Labeled Object Treemap 54
3.14 Adjacency matrices (from [12]) 56
3.15 Colored tree diagrams and Sorted parallel coordinates (from [12]) . 56
3.16 Input for our proposed technique: object tree and taxonomy tree . . 58
3.17 The overall layout of our proposed interactive visualization 59
3.18 All edges option 59
3.19 IDs only option 60
3.20 Visualizing neighbours of similar object type for the object node 10 61
3.21 Visualizing neighbours of different object types for the object node 7 61
3.22 Visualizing all the neighbours of object node 3 62

4.1 Increase(decrease) in USN after adding(removing) a vertex 66
4.2 Lower bound on increase (decrease) in USN after removing (adding) an edge 67
4.3 Upper bound on increase (decrease) in USN after adding (removing) an edge 68
4.4 A valid and optimal labelling of two disjoint copies of K_n using n^2 elements 69
4.5 A) Social Network Graph 71
4.6 B) Corresponding complete graph 72
4.7 Steps for obtaining valid labelling of the Social Network Graph . . 72
4.8 Steps for obtaining valid labelling of the Social Network Graph . . 72
4.9 Edgeless Graph: Our proposed method 74
4.10 Valid labelling of the graph after addition of 2 new vertices and 4 edges 75
4.11 Representation of the graph after addition of 2 new vertices and 4 edges 76
4.12 Proposed method-2 for identify collection of nodes who are mutually non-adjacent 77

5.1 $\text{UUSN}(H) = 6$ 79
5.2 $\text{ILN}(H) = \min\{2, 5, \cdots\} = 2$ 79
5.3 $\text{UUSN}(M_6) = 4$ and $\text{UUSN}(M_6 + v) = 6$ 81
5.4 $\text{ILN}(G) = 1$, $\text{USN}(G) = 3$, and $\text{UUSN}(G) = 3 + 4 * 1 - 3 = 4$ 83
5.5 UUSN labelling- P_7 to P_9 87
5.6 UUSN labelling- P_7 to P_{11} 88
5.7 Input 89
5.8 After Step 1 and 2 90
5.9 After Step 4 91
5.10 Output 92
5.11 Labelling after step 1 93
5.12 Uniform labelling of Q_4 94
5.13 Cartesian Product Based Method 95

6.1 Input graph G 98

6.2 Total graph of G ($T(G)$) 99

6.3 Line graph construction from the given graph 102

6.4 Line graph 102

6.5 Total graph 103

6.6 Degree characteristics of vertex-vertex 104

6.7 Degree characteristics of edge-vertex 104

6.8 Initial steps of direct construction of the total graph of K_3 109

6.9 Resultant Total Graph of K_3 after step 4 109

6.10 Characteristics of Vertex-Vertex and Edge-Vertex 111

7.1 Construction of $Aux(G)$ from G 116

7.2 Idea of Type I and Type II cliques for edge (A.B) 124

7.3 Deletion of the edge e is corresponding to the cycle clique in the $Aux(G)$ 126

7.4 Deletion of the edge e is corresponding to the minimal edge cut clique in the $Aux(G)$ 127

7.5 All edges incident to x are from the same factor 128

CHAPTER 1

Introduction

Using graph theoretic framework, it is possible to solve as well study various problems having practical significance. Graph theory has applications in various fields including computer science (algorithms and computation), electrical engineering (coding theory and communication networks), biochemistry (genomics) and operational research (scheduling). The combinatorial methods of graph theory are also useful in proving fundamental results in other areas of pure mathematics. For an introduction to the basic concepts of graph theory and its applications refer [61] [24] [52].

A graph labelling refers to an assignment of labels to the vertices and/or edges of a graph subject to certain conditions. Formally, given a graph $G = (V, E)$, a vertex labelling is a function of V to a set of labels. A graph with such a function defined is called a vertex-labeled graph. Graph coloring is a special case of graph labelling which is one of the most important problems in graph theory as well as combinatorics. The origin of this problem is the four colour theorem [18] according to which it is possible to colour the regions of a map using four colours and the underlying constraint is regions sharing a boundary in the map must get distinct colours.

In a proper coloring of a graph, assignment of colours is done in such a way that no pair of adjacent vertices get the same colour. Vertices receiving the same colour form an independent set according to this colour assignment scheme. In other words, in a proper coloring, the vertex set is partitioned into independent sets depending on colour classes (vertices having the same colour form one colour class). In general terms, graph coloring refers to the partition of graph elements

(vertices and/or edges) subject to certain constraints.

In this thesis, we consider set labels instead of colours and similar to proper coloring, in our proposed labelling scheme, no pair of adjacent vertices get same (common) element(s) in their corresponding set labels. i.e. set labels of adjacent vertices are disjoint.

Graph labelling techniques were first introduced in mid 1960s. In the intervening 50 years approximately 200 graph labelling techniques have been studied in more than 2000 research papers [20].

1.1 Definitions and notations

In this section, we present some definitions and notations which we use throughout the thesis. The scope of our work is limited to only simple, finite, undirected graphs $G = (V, E)$, where V denotes the set of vertices and E ($E \subseteq V \times V$) denotes the set of edges. We use n to represent $|V|$. We use (a, b) to denote $\{a, b\} \in E$. If $v \in e$, then v is called an endpoint of e (here, $e \in E$). If $(v_1, v_2) \in E$ then we say that vertices v_1 and v_2 are adjacent (or neighbours). If $v \in e$ then the vertex v and the edge e are said to be incident to each other. If an endpoint is shared by edges, then those edges are said to be incident to one other.

Definition 1. *Given $G = (V, E)$ and a vertex $v \in V$, we define the neighbourhood $N(v)$ of v to be the set of neighbours of v. Let the degree $d(v)$ of v be $|N(v)|$, the number of neighbours of v. A vertex v is isolated if $d(v) = 0$.*

Definition 2. *A graph G is d-regular if and only if all vertices have degree d.*

Definition 3. *A graph $H = (U, F)$ is a subgraph of a graph $G = (V, E)$ if $U \subseteq V$and $F \subseteq E$. If $U = V$ then H is called spanning.*

Definition 4. *Given $G = (V, E)$ and $U \subseteq V$ ($U \neq \phi$), let $G[U]$ denote the graph with vertex set U and edge set $E(G[U]) = \{e \in E(G) : e \subseteq U\}$. (We include all the edges of G which have both endpoints in U). Then $G[U]$ is called the subgraph of G induced by U.*

Definition 5. *A k-coloring of G is a labelling $f : V(G) \to \{1, \ldots, k\}$. It is a proper k-coloring if $(x, y) \in E(G)$ implies $f(x) \neq f(y)$. A graph G is k-colourable if it has*

a proper k-coloring. The chromatic number (G) is the minimum k such that G is k-colourable and is denoted by $\chi(G)$.

Definition 6. *An edge of a connected graph is a cut-edge if its deletion disconnects the graph.*

Definition 7. *Given a connected graph G, **a spanning tree** T is a subgraph of G which is a tree and contains every vertex of G.*

Definition 8. *A vertex cut in a connected graph $G = (V, E)$ is a set $S \subseteq V$ such that $G \setminus S := G[V \setminus S]$ has more than one connected component. A cut vertex of a connected graph is a vertex v such that $\{v\}$ is a cut.*

Definition 9. *G is called k-**connected** if $|V(G)| > k$ and if $G \setminus X$ is connected for every set $X \subseteq V$ with $|X| < k$. In other words, no two vertices of G are separated by removal of fewer than k other vertices. Every (non-empty) graph is 0-connected and the 1-connected graphs are precisely the non-trivial connected graphs.*

Definition 10. ***A block** of a graph G is a maximal connected subgraph of G that has no cut-vertex. If G itself is connected and has no cut-vertex, then G is a block. If a block B has at least three vertices, then B is 2-connected. If an edge is a block of G then it is a cut-edge of G.*

Definition 11. *A disconnecting set of edges is a set $F \subseteq E(G)$ such that $G \setminus F$ has more than one component. Given $S, T \subset V(G)$, the notation $[S, T]$ specifies the set of edges having one endpoint in S and the other in T. **An edge cut** is an edge set of the form $[S, \overline{S}]$, where S is a non-empty proper subset of V and $\overline{S} = V - S$.*

1.1.1 Graph classes

In this subsection, we define the graph classes we study.

Definition 12. *K_n is the **complete graph**, or a clique. Take n vertices and all possible edges connecting them.*

Definition 13. *$G = (V, E)$ is **bipartite** if there is a partition $V = V_1 \cup V_2$ into two disjoint sets such that each $e \in E(G)$ intersects both V_1 and V_2.*

Definition 14. *$K_{n,m}$ is the* ***complete bipartite graph****. Take $n+m$ vertices partitioned into a set A of size n and a set B of size m, and include every possible edge between A and B.*

Definition 15. *Given $G=(V,E)$, the* ***complement*** *$\overline{G}$* ***of*** *G has the same vertex set V and $(u,v) \in E(\overline{(G)})$ if and only if $(u,v) \notin E(G)$.*

Definition 16. ***A clique*** *in G is a complete subgraph in G. An independent set is an edgeless induced subgraph in G.*

Definition 17. *A graph having no cycle is acyclic. A forest is an acyclic graph;* ***a tree*** *is a connected acyclic graph. A leaf (or pendant vertex) is a vertex of degree* 1*.*

Definition 18. *A set of edges $M \subseteq E(G)$ in a graph G is called* ***a matching*** *if $e \cap e^{'} = \phi$ for any distinct pair of edges $e, e^{'} \in M$.*

Definition 19. *A* ***path graph*** *has vertices $v_1, v_2 \ldots, v_n$ and edges $e_1, e_2, \ldots, e_{n-1}$, such that each edge e_k joins vertices v_k and v_{k+1}. The path graph on n vertices is denoted by P_n.*

Definition 20. *A* ***cycle graph*** *has vertices $v_1, v_2 \ldots, v_n$ and edges $e_1, e_2, \ldots, e_n$, such that each edge e_k joins vertices v_k and v_{k+1}, for $1 \leq k \leq n-1$ and e_n joins v_n and v_0. The cycle graph on n vertices is denoted C_n.*

Definition 21. *A* ***wheel graph*** *has a hub vertex joined to every other vertex and a cycle through all the other vertices. The wheel graph whose rim is an n-cycle is denoted W_n.*

Some classes of graphs we study are defined on the basis of an operator called the cartesian product. Cartesian product is used to generate complicated graphs from simpler ones. We define this operation below and also define graph classes we consider based on this operator.

Definition 22. *Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the cartesian product $H = G_1 \square G_2$ has vertex set $V = V_1 \times V_2$ where $\times$ represents the cartesian product of the two vertex sets and an edge connects (u_1, u_2) to (v_1, v_2) if and only if $u_1 = v_1$ and $(u_2, v_2) \in E_2$ or $(u_1, v_1) \in E_1$ and $u_2 = v_2$.*

This operator defined for two graphs can be extended iteratively to any number of graphs. The operation is commutative and associative in the sense that the graphs obtained by commuting or bracketing a series of graphs in any order gives rise to the same product graph upto isomorphism. The graph K_1 serves as the identity for the cartesian product operator on graphs.

Definition 23. *A **hypercube** (Q_n) is any graph obtained as the cartesian product of a number of P_2's. Its dimension is the number of P_2's in the product.*

Definition 24. *A **grid or mesh** is any graph obtained as the product of graphs from paths. Its dimension is the number of paths in the product.*

Definition 25. *The* **line graph** $L(G)$ *of a graph* $G = (V, E)$ *is defined as the graph with vertex set having one vertex corresponding to each edge in G and an edge between two vertices of* $L(G)$ *precisely when the edges of G that those vertices correspond to, have a common endpoint.*

Definition 26. *The* **total graph** $T(G)$ *of a graph* $G = (V, E)$ *has as vertex set one vertex for each edge as well as each vertex in G. Two vertices in* $T(G)$ *are adjacent precisely when the elements (vertex or edge) of G they represent are adjacent/incident to each other in G.*

Definition 27. *Given a simple graph G, we define its* **spanning tree auxiliary graph** $Aux(G)$ *as the graph which has a vertex corresponding to each spanning tree of G, and two vertices of* $Aux(G)$ *are adjacent if and only if the corresponding spanning trees in G can be obtained by a single unit transformation.*

1.2 Statements of problem variants

Universe Size Number (USN) of a graph is the number of elements in a smallest universal set S such that one can label the vertices of the graph with unique subsets of S such that if vertices are adjacent then their intersection of labels are disjoint and if the vertices are non-adjacent then their intersection of labels have at least one common element.

Uniform Universe Size Number (UUSN) of a graph is the number of elements

in a smallest universal set S such that one can label the vertices of the graph with unique subsets of S with same cardinality such that if vertices are adjacent then their intersection of labels are disjoint and if the vertices are non-adjacent then their intersection of labels have at least one common element.

Individual Label Number (ILN) of a graph is the smallest size of the largest label over all labellings of the vertices with unique sets such that adjacency coincides with disjointness.

Uniform Individual Label Number (UILN) of a graph is the smallest possible size k for which vertices can be labeled with distinct sets of size k each such that adjacency coincides with disjointness.

1.2.1 Objectives

- To minimize the universe of label used, independent of the individual label sizes.
- To minimize the universe of label used with uniform individual label sizes.
- To minimize the size of individual label size independent of overall universe size.
- To minimize the size of each label in a uniform sized labelling independent of overall universe size.
- Derive relationships (both exact values and upper and/or lower bounds) between these four parameters either universal or specific to some graph classes.
- Derive mathematical results stating bounds or exact values of the size of these labels and the universe set size or constructive methods by providing algorithms for them.
- Characterising total graphs and spanning tree auxiliary graphs.

1.3 Set theoretic definitions of total graphs and spanning tree auxiliary graphs

1.3.1 Total graphs and set labelliing

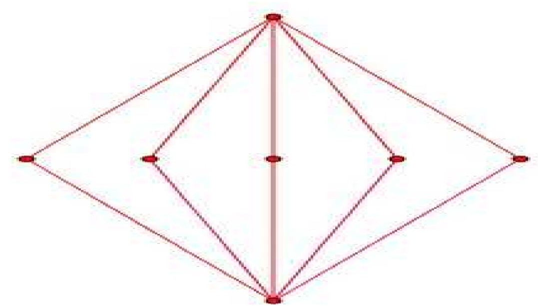


Figure 1.1: Input graph G

Set labelling method to construct the total graph of the given graph:

1. Using $|V|$ elements assign unique singleton label to each vertex of the input graph G.
2. Consider $|E|$ additional vertices. These vertices represent edges of G. For each new vertex, label it using a unique 2-element set, whose elements correspond to the endpoints of the edge which is represented by the newly added vertex.
3. Draw additional edges between pairs of vertices having non-empty singleton intersection between their corresponding labels.
4. The generated graph is the total graph of the given input graph.

An input graph is shown in Figure 1.1 and the corresponding generated total graph is shown in Figure 1.2.

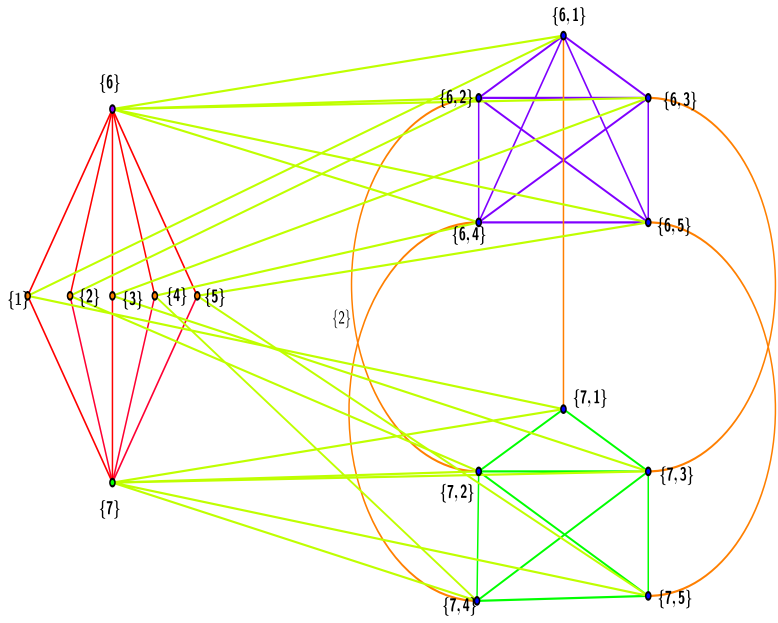


Figure 1.2: Total graph of G ($T(G)$)

1.3.2 Spanning tree auxiliary graph and set labelling

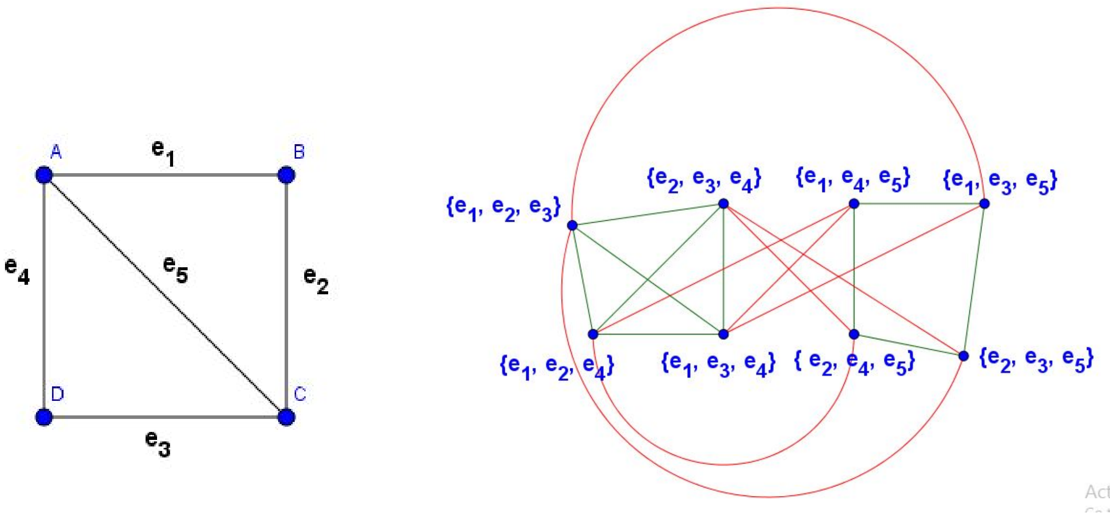


Figure 1.3: Construction of $Aux(G)$ from G

Diagrammatically, one can label the vertices of a simple graph and also its edges with distinct labels. Given such a labelling of a graph G, one can label the vertices of the spanning tree auxiliary graph $Aux(G)$, each with the list of $(n-1)$ edges of the spanning tree it represents. We put an edge between two vertices in the

spanning tree auxiliary graph if and only if the labels of the two vertices share $(n-2)$ of their $(n-1)$ elements in common. See Figure 1.3 for a graph G and its spanning tree auxiliary graph $Aux(G)$.

1.4 Related work

1.4.1 Set labelling and its applications

Knesser graphs $KG_{n,k}$ are graphs whose vertices correspond to the k element subsets of an n element set and two vertices are adjacent precisely when their corresponding subsets are disjoint. Clearly if $n < 2k$ then the graph is an independent set of vertices. If $n = 2k$ then the graph is a matching. When $n = 2k + 1$ we get the special family of **odd graphs**. Knesser graphs are well studied. Many problems on them can be solved clearly and efficiently using this set-theoretic definition. A natural question, therefore, is to try and model an arbitrary graph in this fashion. That is, come up with an underlying universal set and a choice of unique subsets to associate with each vertex such that adjacency is characterised by disjointness of the corresponding subsets. Clearly for an arbitrary graph the above choice of all identical sized subsets of a certain set will not work, because a graph defined in that manner is necessarily vertex transitive.

Our motivation to look at disjointness instead of intersection is that several well known graphs like the Petersen graph and Knesser graphs are expressed in the latter method, and the complements of these families are not well studied. Thus our choice is justified and not merely an attempt to artificially deviate from existing work.

For all the vertices of a graph, it is possible to identify all the neighbours and non-neighbours uniquely using the underlying set labelling. For the given two structurally identical graphs, value of USN/UUSN is same with identical set-labels being used to label vertices. This particular feature is useful to establish isomorphism between two graphs. It is also possible to study ontologies by careful analysis of USN/UUSN and set labels being used for labelling of the graphs.

The closely related concept is **intersection graphs** for finite sets in which non-

adjacency is characterised by disjointness of the corresponding subsets of the underlying universal set. This was studied by Erdos et. al. [14]. In that paper, they also obtain a tight upper bound of $n^2/4$ on the intersection number where the sets are not required to be distinct. In Section 6 of that paper, the authors point out that the problem in general becomes more difficult when the constraint of the distinctness is added. They, however, observe that the universal upper bound applies to that variant as well. We, thus, make inroads into an open problem posed by them obtaining some general results as well as results for some special classes of graphs. We use the slightly different framework of disjointness graphs as many well known families of graphs are defined in this way, as mentioned earlier. So for a given graph, these two labelling approaches are entirely different (except for self-complementary graphs).

For a graph with m edges and n vertices, a trivial upper bound for intersection number is m (see [5]). Alon Noga et. al. [1] derived an upper bound of any n-vertex graph as a function of maximum degree of a graph: $2e^2(d+1)^2 lnn$ where d=maximum degree of the complement graph of G and e=base of the natural logarithm.

Since the problems of intersection and disjointness on graph representation are equivalent, the disjointness version is also NP Complete. The problem of finding a vertex labelling for an arbitrary graph using distinct sets for different vertices and all its standard variants we have listed are NP Complete. This follows from the fact that the equivalent problem of determining the intersection number of an arbitrary graph is NP Complete [14] [22]. Intersection graphs have many applications in the fields of scheduling, biology, VLSI design and they are also used for development of faster algorithms for optimisation problems.

Tree-Maps are used to present hierarchical information on 2-D [58] (or 3-D [11]) displays. Tree-maps offer many features: based on attribute values, users can specify various categories and visualize as well as manipulate categorized information. The traditional approach to represent hierarchical data is to use a directed tree. However, it is impractical to display large (in terms of size as well complexity) trees in limited amount of space. In order to render large trees con-

sisting of millions of nodes efficiently, the Tree-Map algorithm was developed. Even the file system of UNIX can be represented using Tree-Map. The definition of Tree-Maps is recursive: allocate one box for a parent node and the children of that node are drawn as boxes within it. Practically, it is possible to render any tree within a predefined space using this technique. It has applications in many fields including bioinformatics, visualization of stock portfolio.

For many applications it is necessary to consider two aspects: the relationship between different objects and identification of the object type. One can show these aspects using two different trees: 1) taxonomy tree 2) the other tree where each node is related to leaf nodes (object) of a taxonomy. The problem is to design a visualization technique which effectively conveys both the desirable features i.e. relationships between different objects (object tree) along with the mapping of each object with taxonomy. To the best of our knowledge, the best-known visualization technique for representing **multiple hierarchies** is: Trees in a Treemap technique [12].

Graph based information visualization techniques are very well studied in the literature [26] [47] [41] [36]. Applications of Graph Visualization are in many areas. Some of them are: Representation of hierarchical structures using trees, website maps, history of internet browsing data, biology and chemistry (for the representation of phylogenetic trees, genetic maps, molecular maps etc). Other applications include data flow diagrams, entity relationship diagrams, logic programming.

1.4.2 Auxiliary graphs

A total coloring of a simple graph is a simultaneous assignment of colours to its vertices and edges such that adjacent vertices get distinct colours, adjacent edges get distinct colours and the colour of each edge is distinct from the colours of its endpoint vertices (or equivalently the colour of each vertex is distinct from the colours of its incident edges). It is thus a combination of a proper vertex coloring, a proper edge coloring and a further restriction on the interplay between these colorings. The notion of total coloring was introduced by Behzad [9] and Vizing

[60] and those papers also conjectured that $\chi_T(G) \leq \Delta(G) + 2$. It is immediate that $\chi_T(G) \geq \Delta(G) + 1$, since a vertex of maximum degree and its incident edges must all get distinct colours. A lot of work has been done on total coloring [28] [8], based on frugal coloring [27], the list coloring conjecture [51] etc.

The generic concept of **auxiliary graphs** is an important one in graph theory. In its most general form it refers to constructing graphs based on some rules applied to any given graph. In other words it is a function from the set of graphs to the set of graphs. The definition is usually based on some natural and important properties of graphs and the computation of the function is easy in principle, even if the algorithm involved could be expensive in terms of computational complexity. What is usually less clear is the range of this function. It is rare for the range of these auxiliary functions to be the entire codomain (in this case the class of all graphs). Thus the challenging and important problems associated with these auxiliary graph families is to characterise mathematical properties of graphs which belong to the range, algorithms for deciding whether a graph belongs to the range or not and also algorithms for computing the inverse image of a given auxiliary graph if it is unique. If the preimage is not unique, then one interesting challenge is to decide what constitutes a minimal/canonical solution and also ways to generate the entire set of solutions by extending the basic solutions.

A well known example of auxiliary graphs is the class of line graphs. The characterisation as well as algorithms for recognising this class of graphs and computing their inverse images has been established in a wide variety of research articles [7] [43] [54].

1.5 Thesis outline

In Chapter 2, we present upper bounds/optimal values on USN for some special families of graphs including paths, cycles, k-ary trees, complete bipartite graphs, matching, complement of complete graphs. Chapter 3 explains treemaps as well as our proposed visualization technique for multiple hierarchies namely 'Labeled Object Treemap'. The implemented interactive version of the proposed method is

also explained in the same chapter. Our other proposed graph visualization technique ('Edgeless graph') is explained in Chapter 4. In Chapter 5, we present upper bounds/optimal values on UUSN and ILN for some special families of graphs. The next two chapters, Chapter 6 and Chapter 7 give new characterisations of total graphs and spanning tree auxiliary graphs respectively. Chapter 8 contains some concluding remarks and outlines possible future directions for research.

Chapter 2

Results on Universe Size Number (USN)

In Section 2.1, the problem statement is discussed along with some basic results. Results related to USN are discussed in detail for some specific classes of graphs (including complement of complete graph, matching, paths, cycles, complete bipartite graph) in Section 2.2. Cartesian product based method is discussed in Section 2.3. In Section 2.4, results related to USN of complete binary trees and k-ary trees are presented. A modified algorithm to quickly identify non-neighbours as well as neighbours of a particular node of the complete binary tree is also explained in the same section. The final section summarises the results obtained in this chapter.

2.1 Introduction

A set labelling of a graph $G(V, E)$ is a function $f : V \to \mathcal{P}(\{1, 2, \ldots, k\})$ where $k \in Z^+$ such that

- f is one one.
- $\forall u, v \in V, (u, v) \in E \Leftrightarrow f(u) \cap f(v) = \phi$.

Universe size number (USN) of a graph is the least positive integer k such that a set labelling of G exists. (For example, see Figure 2.1)

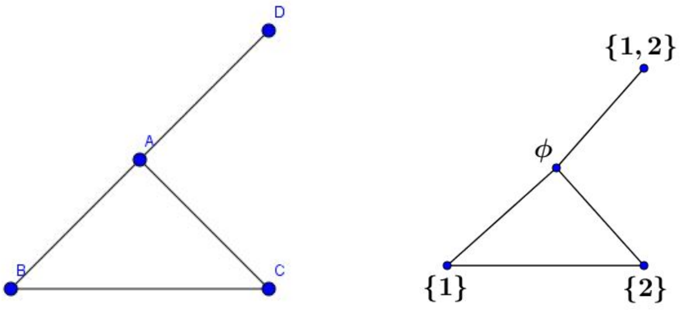


Figure 2.1: Input graph H and $\text{USN}(H) = 2$

2.1.1 General results on USN

Here, we present general bounds on USN.

Theorem 2.1.1. *$\text{USN}(G) \geq \lceil log_2 n \rceil$ where G has n vertices.*

Proof. Case 1: $n = 2^i$ where $i \in N$. Using $\lceil \log_2 n \rceil$ elements, at most n subsets can be generated which, due to the requirement of label distinctness, can be used to label at most n vertices. This proves the claim.
Case 2: $n \neq 2^i$ where $i \in N$. Using $\lfloor \log_2 n \rfloor$ elements, at most $n-1$ subsets can be generated which, due to the requirement of label distinctness, can be used to label at most $n-1$ vertices. Here total number of vertices is n. In order to assign a non-empty as well as unique label to n^{th} vertex 1 additional element is required in the underlying universe.
Therefore value of USN is at least $\lceil \log_2 n \rceil$ for any given graph. □

Theorem 2.1.2. *$\text{USN}(G + v) \leq \text{USN}(G) + 1$, if $d(v) = n(G)$.*

Proof. Since the vertex v is adjacent to all other vertices, no element used in its label can be used in the labels of any other vertex. Disregarding v, the graph G requires $\text{USN}(G)$ elements for labelling its vertices even as a subgraph of G'. If ϕ is not used in the labelling of vertices of G then we can assign ϕ as a valid label to the newly added vertex v. In this case, $\text{USN}(G + v) = \text{USN}(G)$. In all other cases, one additional element is required to label the vertex v. There is no purpose served in having more than one element in the label of v because v has no non-neighbours. □

Theorem 2.1.3. $\text{USN}(K_n) = n - 1$

Proof. Label the n vertices of K_n with sets $\{\phi, \{1\}, \{2\}, \{3\}, \ldots, \{n-1\}\}$. This assignment respects adjacency as well as non-adjacency among all pairs of vertices. The underlying labelling is valid and it uses $n - 1$ elements. Hence $\text{USN}(K_n) \leq n - 1$. In the case of K_n, each vertex is adjacent to all the remaining vertices and hence it is not possible to reuse any of the existing individual vertex label element(s). Using $n - 1$ elements, it is possible to generate at most n distinct sets having empty intersection between any two sets. Hence $\text{USN}(K_n) \geq n - 1$ which proves the result. □

2.2 Results on USN for some special families of graphs

In this section, we derive results on USN (either exact or asymptotic) on the following classes of graphs: Paths, Cycles, Wheel Graph, Hypercube, Complete Graph, Complement of Complete Graph $(\overline{K_n})$, Matching (M_{2n})).

Theorem 2.2.1. *For matching,* $\text{USN} = O(\log_2 2n)$

Proof. Consider matching graph on $2n$ vertices (number of edges=n). In a matching each vertex of the graph is adjacent to exactly one other vertex. Hence, the label of each vertex must have non-empty intersection with the labels of all the remaining $2n - 2$ vertices. Assume that the underlying universal set U has k elements. Our aim is to calculate the value of k. We can summarize our requirements:

1. k should be large enough to generate at least $2n$ non-empty subsets.
2. Each subset must have non-empty intersection with all subsets except its unique neighbor.

If we consider only subsets of cardinality $k/2$ then each subset will have exactly one disjoint subset and it will have non-empty intersection with all the remaining subsets (pigeonhole principle). In order to fulfill the first requirement, $\binom{k}{k/2} \geq 2n$. By Stirling's approximation, USN=k=$O(\log_2 2n)$

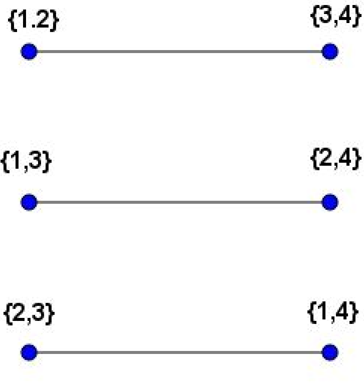


Figure 2.2: USN $(M_6) = 4$

□

For example, USN $(M_6) = 4$ (See Figure 2.2).

Theorem 2.2.2. USN$(\overline{K_n}) = 1 + \lceil \log_2 n \rceil$

Proof. Consider an underlying universal set A with cardinality k.
Disjoint sets are not allowed for labelling distinct vertices of an independent set. For all S (where $S \subset A$), at most one subset can be used for labelling from each pair $(S, A - S)$. So the total available sets for labelling are reduced by factor of half. Select the subset having higher cardinality in each pair and if cardinalities are same then make an arbitary selection. By doing this, each subset will have cardinality at least $\frac{k}{2}$. All these selected subsets will have non-empty intersection (pigenhole principle) and hence they can be used to assign labels to vertices.
So, in summary, at most 2^{k-1} vertices of the graph can be labelled using A. Similarly, at most 2^{k-2} vertices of the graph can be labelled using $k-1$ labels. So using k labels, it is possible to get a valid and optimal labelling of $(\overline{K_n})$ where $n \in \{2^{k-2}+1, 2^{k-2}+2, \ldots, 2^{k-1}\}$ which proves the result.

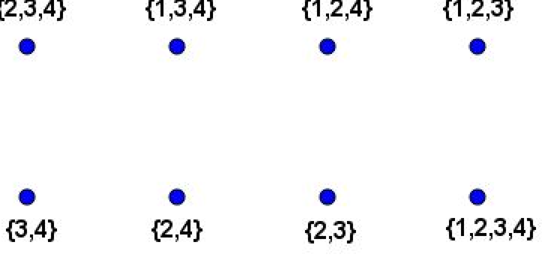


Figure 2.3: USN $(\overline{K_8}) = 4$

□

For example, USN $(\overline{K_8}) = 4$ (see Figure 2.3).

Theorem 2.2.3. USN $K_{s,t} = 2 + \lceil \log_2 s \rceil + \lceil \log_2 t \rceil$

Proof. Complete bipartite graphs consist of two independent sets. From the valid and optimal labelling of one independent set nothing can be used in the second independent set. So labelling of these two independent sets must be entirely disjoint which proves the result. □

Theorem 2.2.4. USN$(P_n) = O(\log n)$.

Add-edge Procedure:

Input: A valid labelling of any given path (P_n) using exactly k labels. i.e. USN$(P_n) \leq k$.

Output: A valid labelling of P_{n+2} using exactly $k+3$ labels. i.e. USN$(P_n) \leq k+3$.

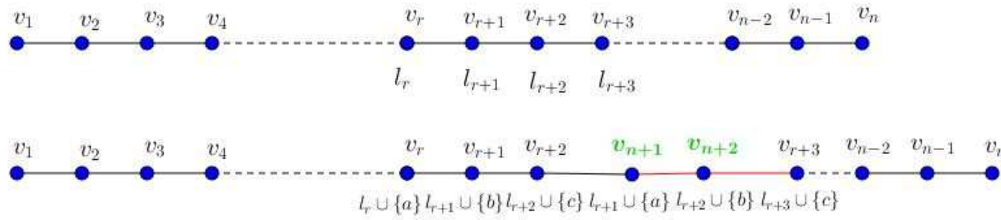


Figure 2.4: Summary of add-edge procedure

Step 1: Identify a P_4 in the given P_n. Let the vertices of this P_4 be v_r, v_{r+1}, v_{r+2} and v_{r+3} and corresponding labels be l_r, l_{r+1}, l_{r+2} and l_{r+3} respectively.

Step 2: Add 2 new vertices v_{n+1} and v_{n+2}. In order to construct P_{n+2}, add 3 new edges (v_{r+2}, v_{n+1}), (v_{n+1}, v_{n+2}), (v_{n+2}, v_{r+3}).

Step 3: $l(v_{n+1}) = l_{r+1}$

$l(v_{n+2}) = l_{r+2}$

Step 4: Three pairs (v_r, v_{n+1}), (v_{r+1}, v_{n+2}) and (v_{r+2}, v_{r+3}) are non-adjacent. In order to reflect the non-adjacency in the labelling, add 3 distinct new elements say a, b and c to the labels of these 3 pairs respectively. The final labelling is valid. It respects adjacency as well as non-adjacency for all pairs of vertices.

A summary of the Add-edge procedure is shown in Figure 2.4.

Lemma 2.2.1. *For the given P_n, the add-edge procedure can be applied for at most $\frac{n-1}{3}$.*

Proof. The Add-edge procedure can only be applied to each edge-disjoint P_4-subgraph of the given P_n. The maximum number of edge-disjoint P_4 in a P_n is $\frac{n-1}{3}$. Thus the result follows. □

Proof of Theorem 2.2.4:

We now give an algorithmic proof.

Algorithm: ValidPathLabelling:

Input: A valid labelling of P_n using exactly k labels.

Output: A valid labelling of P_{n+2i} using exactly $k+3$ labels where $0 < i \leq \frac{n-1}{3}$ and $i \in N^+$

Step 1: Apply the Add-edge procedure on the given P_n for the first 4 vertices i.e. $r = 1$.

Step 2: For $j \geq 1$, (where $j \in N^+$)

If $3j+1 < n$ and $3j+4 \leq n$ then apply the Add-edge procedure for $r = 3j+1$ with the following modifications in step 4 of the procedure:

(Observation: v_{3j+1} participates in exactly two Add-edge procedures; in iteration $j-1$ as well as iteration j. Therefore, before starting of iteration j, the label of v_{3j+1} already has an additional element p due to iteration $j-1$ where $p \in \{a, b, c\}$ For $j = 1$, $p = \{c\}$ from step 1. The label of v_{3j+1} won't change during the jth iteration and $p \in l_{3j+1}$.)

1. Add p in the label of newly added vertex v which is a neighbour of v_{3j+3}.

2. Use the remaining two elements $\{a, b, c\} - \{p\}$ in any order, for the remaining two pairs.

The final labelling of P_{n+2i} is valid after applying valid-path-labelling algorithm.

Upper bound calculation:

So at most $\left\lfloor \frac{n-1}{3} \right\rfloor$ edges can be used and they will generate at most $\dfrac{2n-2}{3}$ new vertices.

Recurrence: $T(5n/3) = T(n) + 3$

USN : $O(\log n)$ $\square$

Note: We can generate paths for all values of n by applying the Add-edge procedure repeatedly. One can generate all P_{2x+1} by repeatedly applying the add-edge procedure on a valid labelling of P_7, similarly all P_{2x} can be generated by application of the add-edge procedure on a valid and optimal labelling of P_6. Here $x > 3$ and $x \in \mathcal{N}$.

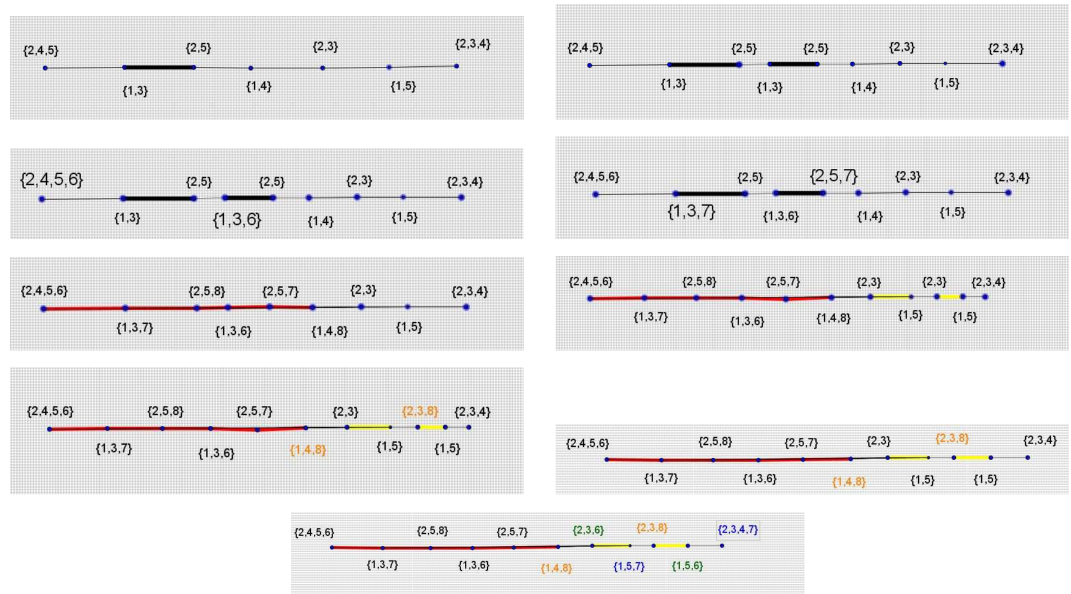


Figure 2.5: Add-edge procedure : P_7 to P_{11}

Construction of P_{11} from P_7 is shown in Figure 2.5.

As a corollary we have the following theorem.

Theorem 2.2.5. $\text{USN}(C_n) = O(\log n)$ *and* $\text{USN}(W_n) = O(\log n)$.

Proof. With the use of the ValidPathLabelling algorithm, it is possible to generate a valid labelling of C_n using $O(\log n)$ labels. Notice that the edge (V_1, V_n) does not participate in any of the Add-edge procedure iterations. Wheel graph consists of C_n and one additional vertex with degree n. Therefore, $\text{USN}(W_n) = \text{USN}(C_n)$ (from Theorem 2.1.2). □

Theorem 2.2.6. $\text{USN}(Q_n) < 3n + O(logn)$

For hypercube, the valid labelling problem can be viewed as two independent subproblems:

1. Valid labelling of levels in order to reflect non-adjacency between them (except adjacent levels- if at least one edge is present between two levels then those two levels are adjacent), non-adjacent levels should have at least one element in common to reflect non-adjacency.

2. Valid labelling to reflect adjacency and non-adjacency between specific pairs of vertices in adjacent levels.

Traditional hasse diagrams type labellings for Q_4 are shown in Figures 2.6 and 2.7.

Proof. We will give an algorithmic proof for the claim.

Algorithm:

Input: Q_n

Output: A valid labelling of Q_n with at most $3n + O(\log n)$ labels.

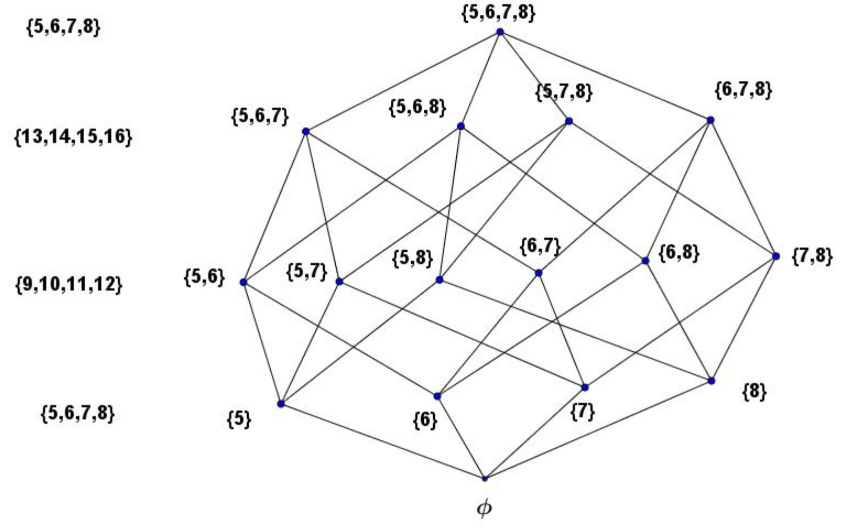


Figure 2.6: Hasse diagram type labelling of Q_4 using the set {5,6,7,8}

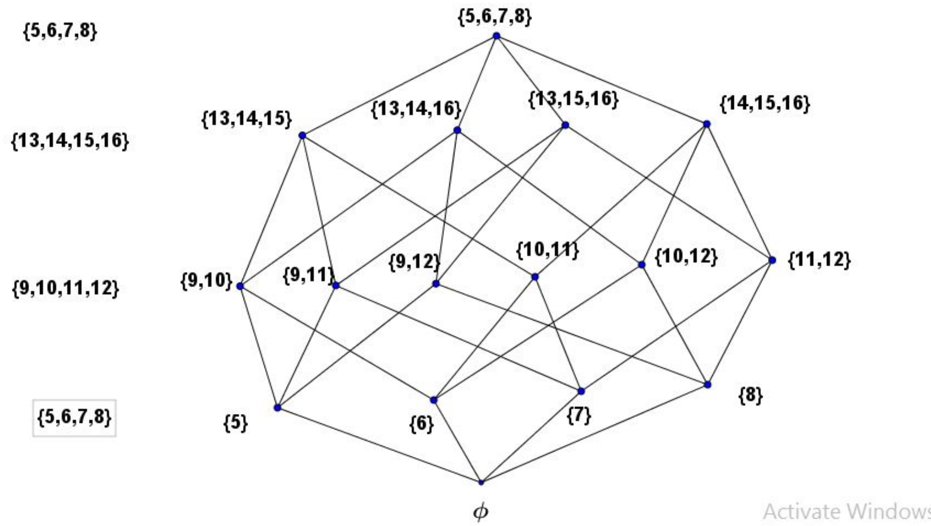


Figure 2.7: Hasse diagram type labelling of Q_4 using 3 disjoint sets with the same cardinality (4)

Step 1:

Consider P_{n+1} (path of length n) for given Q_n. Apply ValidPathLabelling algorithm to get a valid labelling of P_{n+1} with $O(\log n)$ labels. Now, for all i, labels of all vertices at level i= label of vertex P_{i+1}.

For example, for Q_6 consider a valid and optimal labelling of P_7 and assign 245, 13, 25, 14, 23, 15, 234 to the labels of the vertices of the corresponding 7 levels of Q_6. Valid labelling of P_5 and the corresponding 5 levels of Q_4 are shown in Figure 2.8.

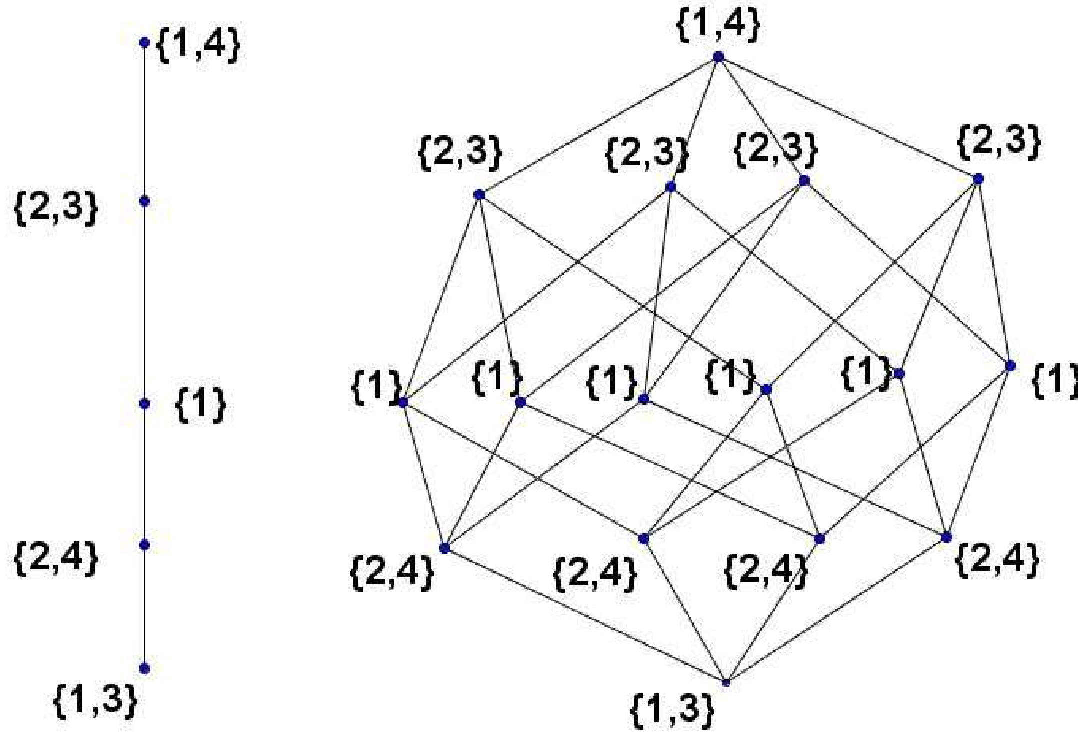


Figure 2.8: Step 1

So far, $O(\log n)$ element have been used for the labelling and after this step all the vertices of non-adjacent levels reflect non-adjacency.
Additionally all vertices within one level have at least one common element. So non-adjacency within one level is respected too.

Step 2 (a)

Consider three disjoint (disjoint with elements used in Step 1 of labelling) sets each of size n: S_1, S_2 and S_3.
S_1 is used for labelling of levels: $3, 6, 9, \dots$
Similarly S_2 is used for labelling of levels: $1, 4, 7, 10, \dots$
and S_3 is used for labelling of levels: $2, 5, 8, 11, \dots$
Q_n contains $\binom{n}{i}$ vertices at level i. Assign $\binom{n}{i}$ different labels to all vertices of level i using the $\binom{n}{i}$ subsets of the underlying set S mentioned above. (Using conventional subset assignment method to hypercube i.e. level i should contain all subsets of length i).
So after this step $2(a)$, all labels are unique.

Note: For the 0th level use valid labelling of 1st vertex of P_n obtained from step 1. After step 2(a), adjacency is preserved fully because all adjacent vertices have disjoint labels because they use disjoint underlying sets for labelling (see Figure 2.9). Note: The labelling shown in Figure 2.9 is the union of labellings shown in Figures 2.7 and 2.8.

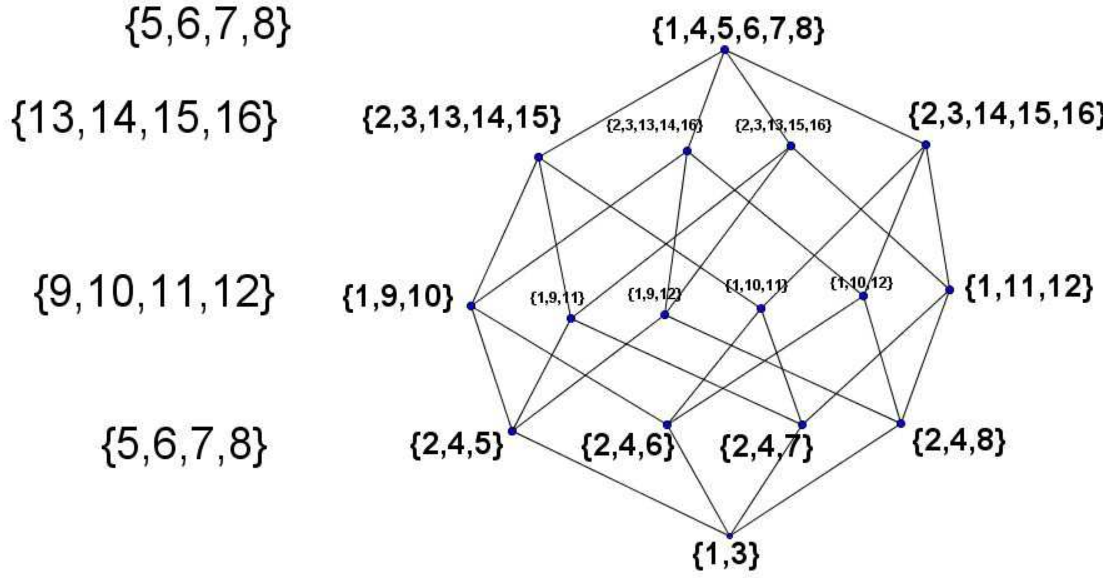


Figure 2.9: Step $2(a)$

A total of $3n + O(\log n)$ labels are used so far.

Step 2(b):

Now non-adjacency between two adjacent levels remains to be addressed. For each vertex of level i, add extra elements from level $i-1$ to preserve non-adjacency with level $i-1$.

Procedure to add extra elements to the label of a vertex v of level i :

1. Consider all vertices which are adjacent to vertex v and at level $i-1$. Take the union of all labels of these vertices, say P.

2. Let $Q = S \setminus P$ where $S : S_1/S_2/S_3$ which is used to label vertices of level $i-1$.

3. New label of v= {old label of v}$\cup Q$.

Now the ith level will contain elements from both sets: set used for level i and $i-1$.

Add extra elements only from the originally assigned underlying set in $2(a)$. i.e. for level $i+1$ add extra elements to preserve non-adjacency from the underlying set of level i not $i-1$.

No new labels are added in this step and hence total number of labels used after all steps is $3n + O(\log n)$. Final labelling after step $2(b)$ preserves adjacency as well as non-adjacency for all pairs of vertices. $\square$

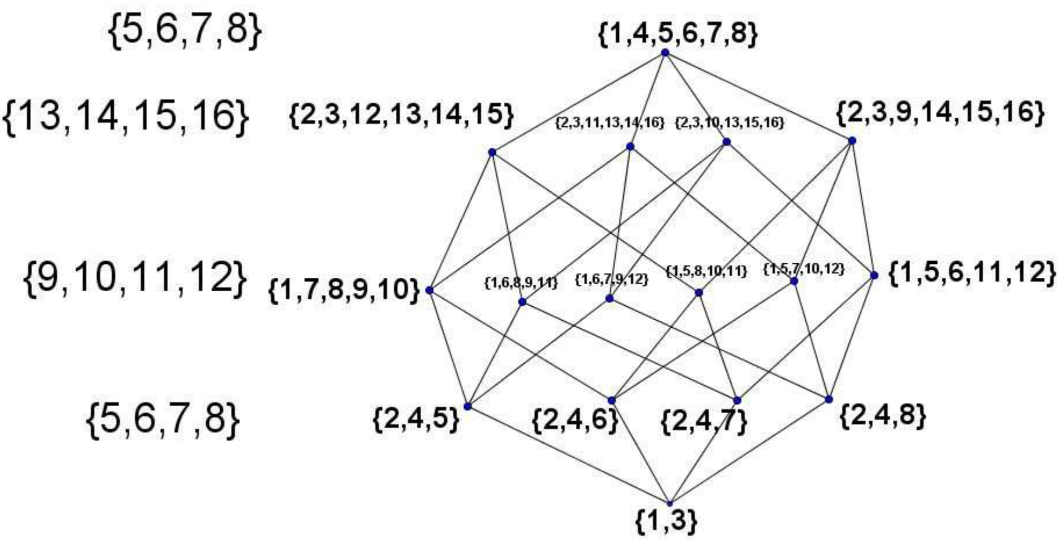


Figure 2.10: Final labelling after Step $2b$

Final labelling for Q_4 is shown in Figure 5.4.

2.2.1 Grid graph and ladder graph

For the given $G_{m,n}$ grid, assume the valid labelling for $m = 1$ to $m = i$ is done. How to construct valid labelling for $m = i + 1$ is explained in the Figures 2.11 and 2.12.

Valid-Grid-Labelling:

Step 1: As shown in the Figure 2.11, $V_{i,k}$ and $V_{i+1,k-1}$ are structurally highly similar. For $k \geq 2$, $L(V_{i+1,k-1}) = L(V_{i,k})$.

Note: The last vertex namely $V_{i+1,n}$ remains unlabelled after this step.

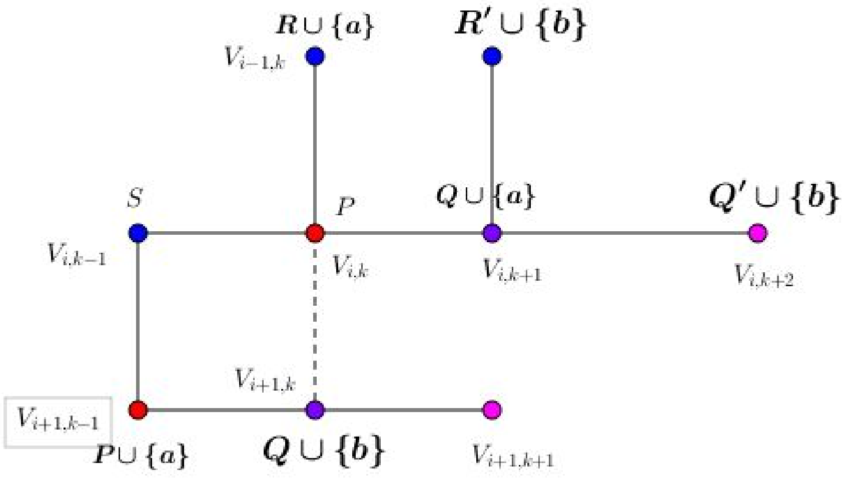


Figure 2.11: Structrual similarity of $V_{i,k}$ and $V_{i+1,k-1}$

Step 2: For $i \geq 2$, 2 vertices, $V_{i,k+1}$ and $V_{i-1,k}$ are non-adjacent to $V_{i+1,k-1}$ but adjacent to $V_{i,k}$. Hence, add a common element a in the labels of $V_{i,k+1}$, $V_{i-1,k}$ and $V_{i-1,k-1}$.

The need for 3 new elements is explained in the Figure 2.12. In general, for $m = i + 1$, for all $V_{m,k}$, use a iff $k \mod 3 = 1$, use b iff $k \mod 3 = 2$ and use c iff $k \mod 3 = 0$ in order to satisfy non-adjacency with respect to two vertices of $m = i$ and $m = i - 1$.

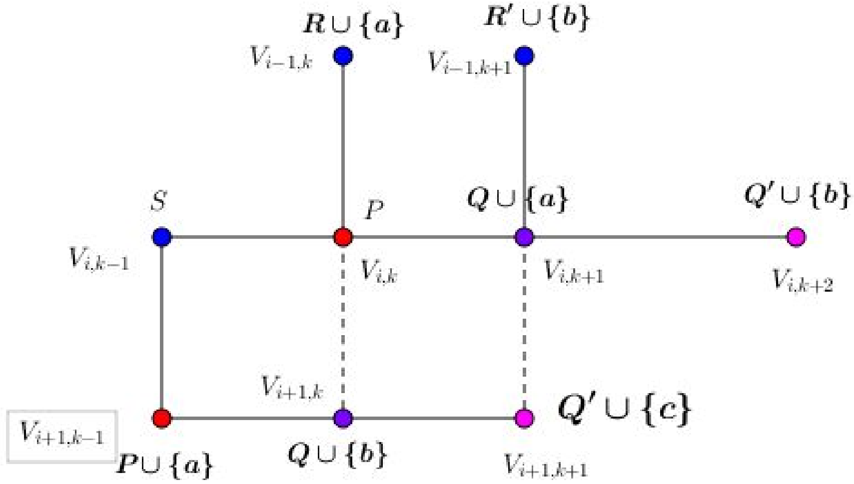


Figure 2.12: Justification of 3rd new element

Remaining steps are for the labelling of $V_{i+1,n}$.

Step 3: Identify all vertices V which are at distance 2 from $V_{i+1,n}$.

Say A= Intersection of the labels of all vertices at distance 2 from $V_{i+2,n}$. Assign $L(V_{i+1,n}) = A$.

Step 4: Let B= All vertices V which are at distance 3 from $V_{i+1,n}$.

Assign $L(V_{i+1,n}) = A \cup \{d\}$.

Add $\{d\}$ to all labels of B.

Theorem 2.2.7. *For the Ladder Graph,* $\text{USN}(L_n) = O(logn)$

Proof. Base case is shown in the Figure 2.13 and final valid labelling can be obtained by applying Valid-Grid-Labelling algorithm with appropriate modifications. Construction steps are shown in the Figures 2.14- 2.16. □

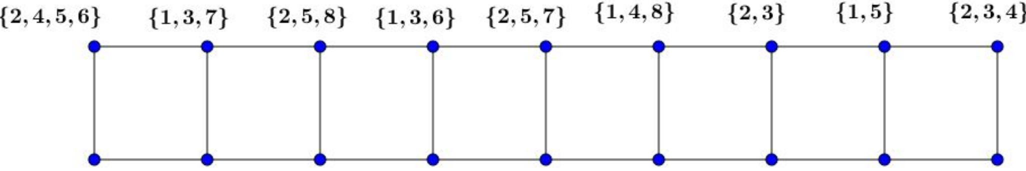


Figure 2.13: Base case

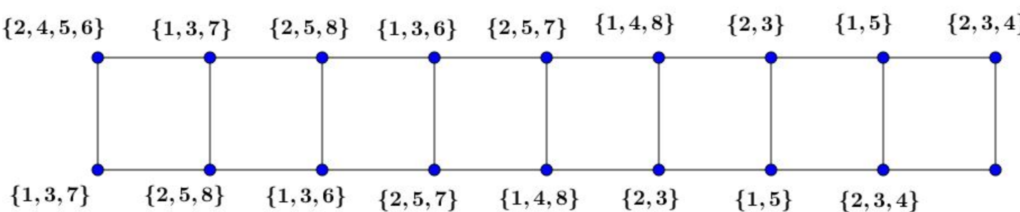


Figure 2.14: Step 1

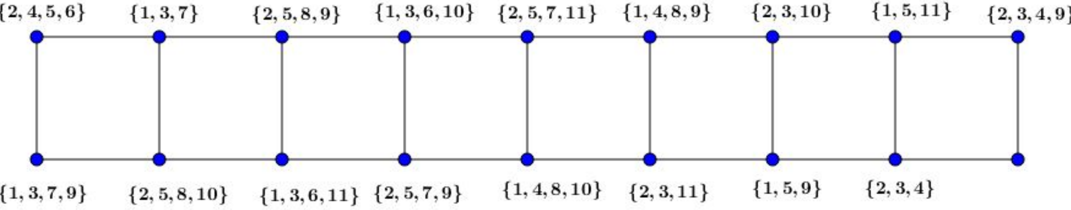


Figure 2.15: Step 2

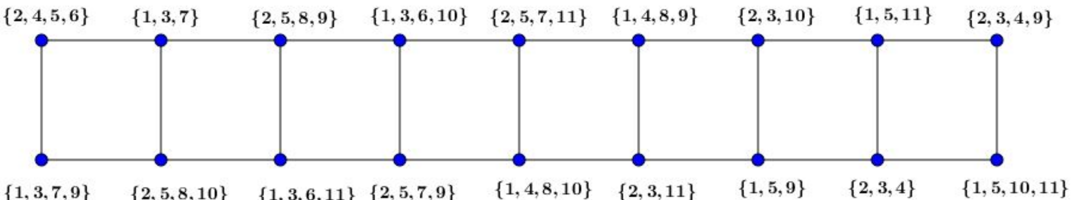


Figure 2.16: Final labelling

Theorem 2.2.8. $\text{USN}(G_{m,n}) = O(\sqrt{p})$ *where* $p = m * n$

Proof. Using Valid-Path-Labelling algorithm, it is possible to label n vertices for $m = 1$. Remaining vertices can be labelled using Valid-Grid-Labelling algorithm. For each value of m, at most 4 extra elements are required and hence $\text{USN}(G_{m,n}) = 4m + O(logn)$. □

Construction of valid labelling of $G_{3,9}$ as well as $G_{4,9}$ are shown below.

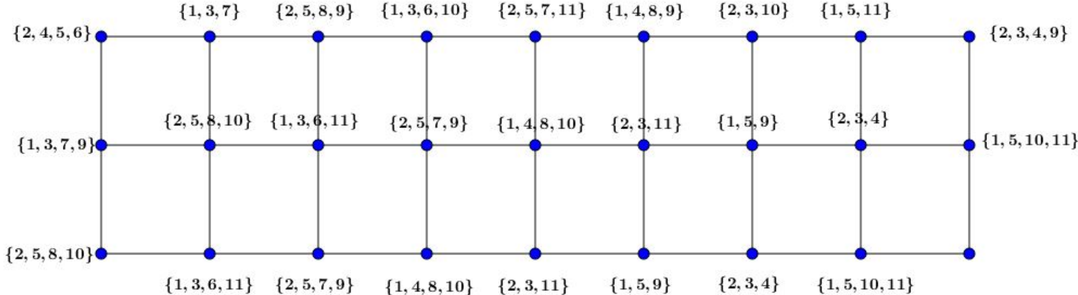


Figure 2.17: $G_{3,9}$ after step 1

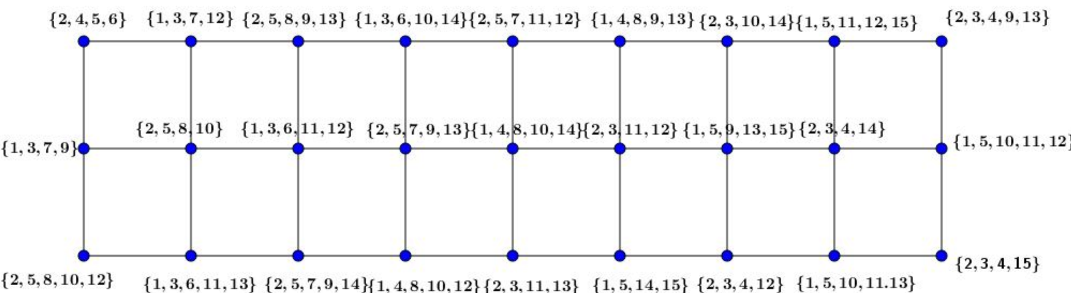


Figure 2.18: Final labelling $G_{3,9}$

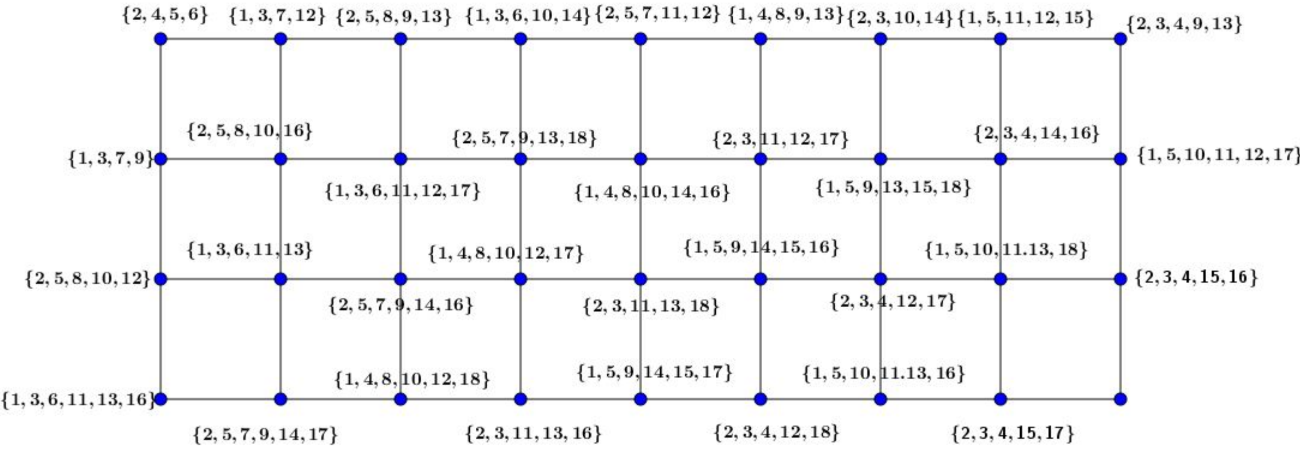


Figure 2.19: $G_{4,9}$ after steps 1 and 2

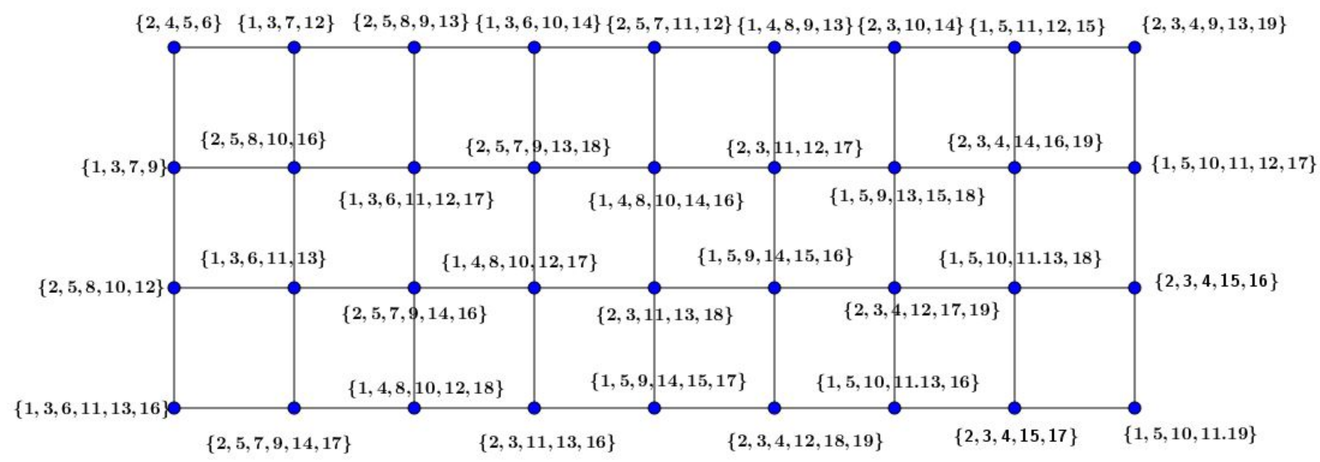


Figure 2.20: Final labelling $G_{4,9}$

2.3 Cartesian product based method

Key Observation: Let A, B, C, D be sets. Then $(A \times B) \cap (C \times D) \neq \emptyset$ if and only if $A \cap C \neq \emptyset$ and $B \cap D \neq \emptyset$.

Theorem 2.3.1. *Let G and H be two graphs on the same vertex set V. Further, suppose $E(G) \cap E(H) = \emptyset$. Then* $\text{USN}(G + H) \leq \text{USN}(G) \times \text{USN}(H)$.

Proof. Take optimal labellings of the vertex set using disjoint universes for the graphs G and H. Now consider the graph $G + H$. The vertex sets of G and H

are identical. For each vertex v in $G + H$ give it the label $l_G(v) \times l_H(v)$. Clearly two vertices are nonadjacent only if they are nonadjacent in both G and H. In that case their labels under the two labellings will each be intersecting. From the key observation, it follows that their cartesian product new label will also intersect. Similarly for the case of non-intersection (adjacent vertices). □

As a corollary we have the following theorem.

Theorem 2.3.2. $\text{USN}(P_n) \leq (1 + \log \frac{n}{2})^2$.

Proof. The path is the union of two disjoint matchings. Each matching has USN $O(\log n)$. From Theorem 2.3.1, we see that the graph has USN $O(\log n)^2$. We state this here just as an application since we have a better bound for paths (Theorem 2.2.4). □

By appling Theorem 2.3.2, valid labelling of P_6 is shown in Figure 2.21.

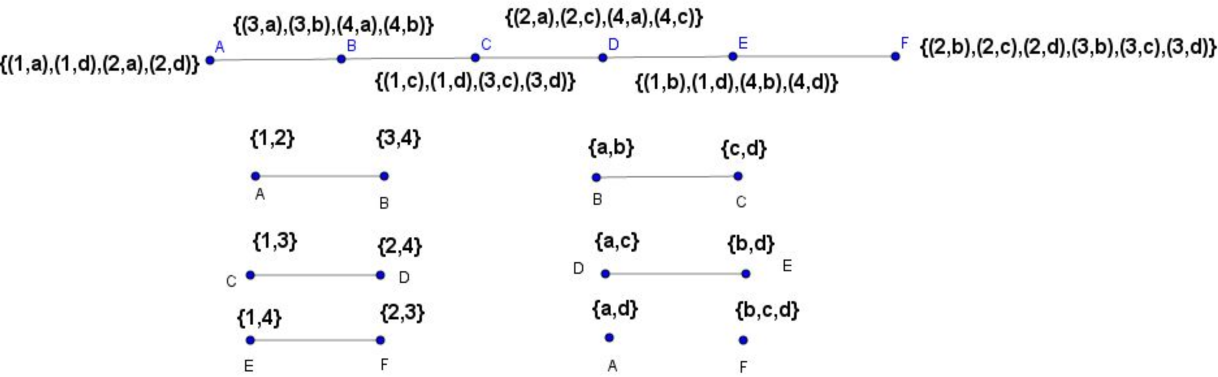


Figure 2.21: Cartesian Product Based Method

2.4 Results on USN for complete k-ary trees

In this section, we derive results on USN (asymptotic) on the following classes of graphs: complete binary trees and complete k-ary trees. Structural similarities between level L_{N+1} and L_{N-1} of complete binary tree is highlighted in Figure 2.22.

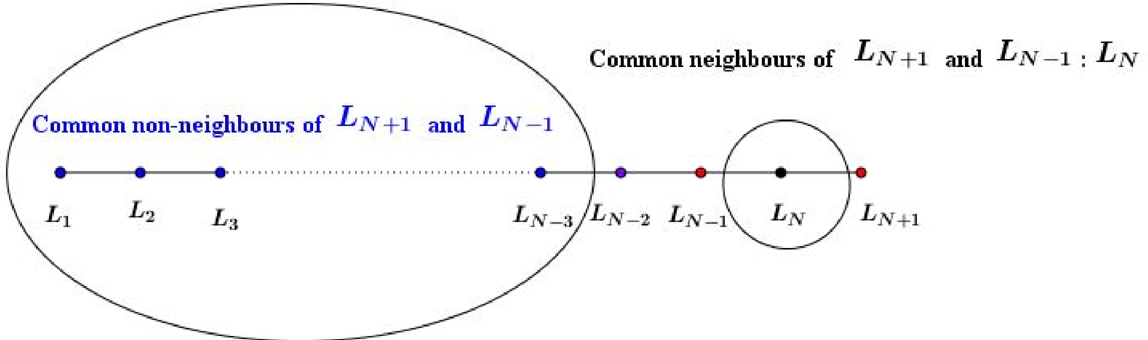


Figure 2.22: Summary of adjacency and non-adjacency for L_{N+1} and L_{N-1}

Theorem 2.4.1. $\text{USN}(BT_n) = O(logn)$. *Where BT=Complete binary tree and n denotes the total number of vertices of the BT.*

We prove this result by an iterative algorithm: Valid-BinaryTree-Labelling. The algorithm is explained below:
Input: A valid labelling of the complete binary tree of height h using exactly K labels.
Base case: $\text{USN}(BT_7) = 5$. Base case is shown in Fig. 2.23 with underlying labelling set: $\{1, 2, 3, 4, 5\}$.

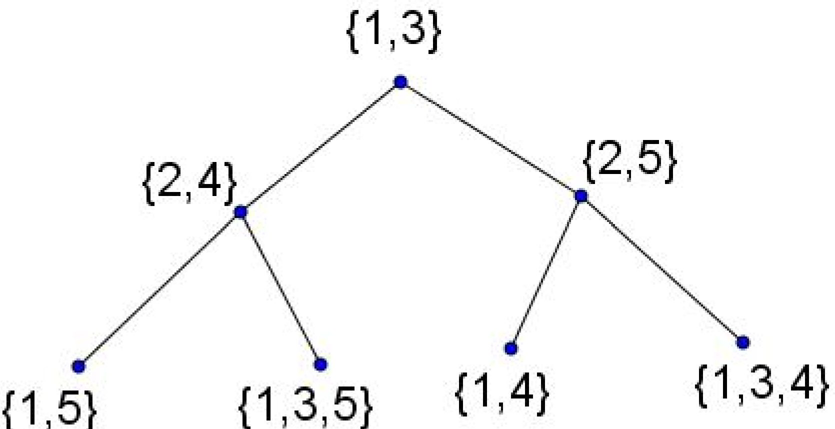


Figure 2.23: Input

Output: A valid labelling of complete binary tree of height $h+1$ using exactly $K+4$ labels. The four new labels are a,b,c and d.

Note: During the phase of the algorithm when the number of levels increases from L to $L+1$, labels of vertices which are present at level $L-2$, L and $L+1$ are changed. Labels of all other vertices remain as it is.

Step 1: For all newly added vertices in level $L+1$, find their corresponding ancestors in level $L-1$.

For all v (where v is a level $L+1$ vertex) ,

Label(v)= Label(ancestor(v) in level $L-1$).

Note: The levels L+1 and L-1 are highly similar with respect to adjacency with the remaining levels. Both of these levels are adjacent to level L and non-adjacent to $1, 2, 3, \ldots, L-3$.

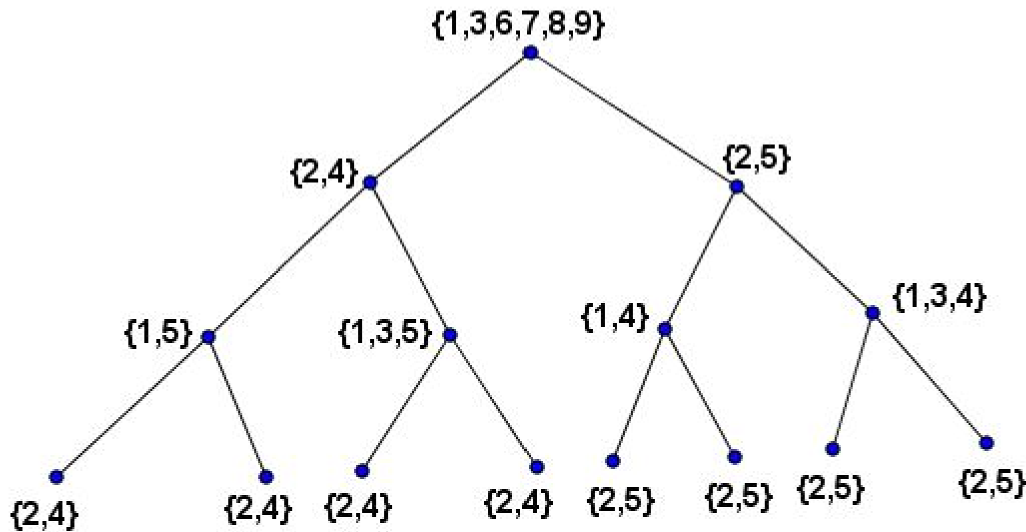


Figure 2.24: After Step 1 and 2

Observations after step 1:

1. All the vertex-labels which are present in level $L+1$ have non-empty intersection with each other because corresponding ancestors are either same or having non-empty intersection. All vertices present at level $L-2$ form an independent set and since the underlying labelling is valid, all the vertex-labels pairs have non-empty intersection. So the vertex-labels which are present in level $L+1$ have

non-empty intersection with each other because they are generated from the vertex labels present at $L-2$. This is desirable because all vertices of level $L+1$ are non adjacent.
2. Layer $L+1$ and $L-1$ are non adjacent and they have non-empty intersection after this step.

Old-Label(v) in step i represents a label of v obtained after steps 1 to $i-1$.
New-Label(v) in step i represents a new(modified) label of v after step i.

Step 2: The level $L-2$ is adjacent to $L-1$ but not to $L+1$.
For all v^i, (where v' is a level $L-2$ vertex)
$New\text{-}Label(v') = Old\text{-}Label(v') \cup \{a,b,c,d\}$ (See Figure 2.24)

Step 3: Consider the sequential ordering (left to right) of vertices which are present in level $L+1$.

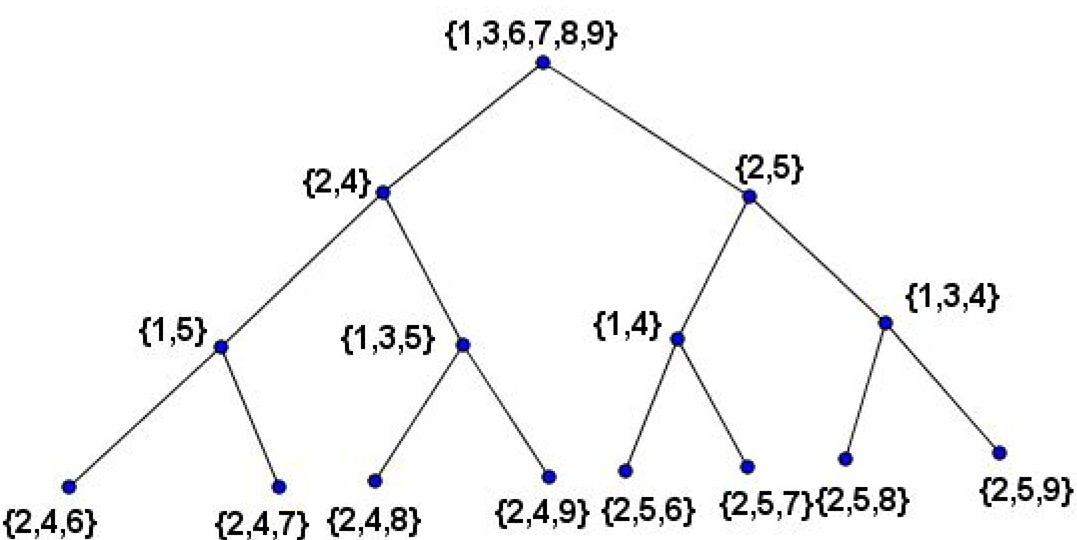


Figure 2.25: After Step 3

$$\begin{aligned} \text{For each vertex } v_i,\ New\text{-}Label(v_i) &= Old\text{-}Label(v_i) \cup \{a\} \text{ if } i \mod 4 = 1 \\ &= Old\text{-}\ Label(v_i) \cup \{b\} \text{ if } i \mod 4 = 2 \\ &= Old\text{-}\ Label(v_i) \cup \{c\} \text{ if } i \mod 4 = 3 \end{aligned}$$

$= Old\text{-}Label(v_i) \cup \{d\}$ if $i \mod 4 = 0$

Vertices of level $L-2$ and $L+1$ will preserve non-adjacency because the corresponding labels have non-empty intersection after this step (see Figure 2.25). The reason for using 4 distinct elements is to ensure uniqueness at Level $L+1$.

Step 4: Consider the sequential ordering (left to right) of vertices which are present in level $L : v_1, v_2, \ldots, v_q$

For each vertex v_i,

$New\text{-}Label(v_i) = Old\text{-}Label(v_i) \cup \{c, d\}$ if $i \mod 2 = 1$

$= Old\text{-} Label(v_i) \cup \{a, b\}$ if $i \mod 2 = 0$

Observation after step 4:

Level $L+1$ and L preserves adjacency as well as non-adjacency.

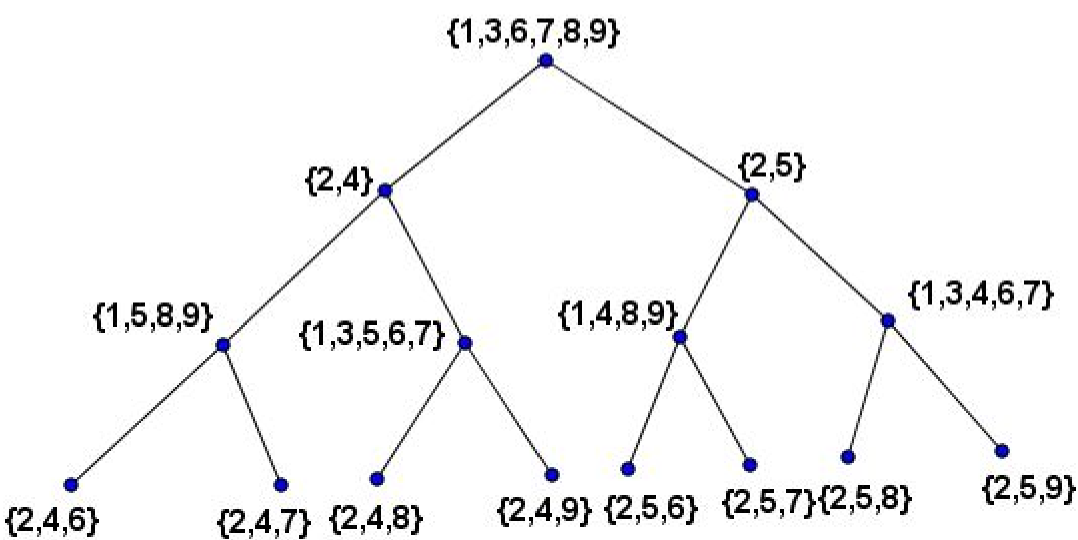


Figure 2.26: Output

The final labelling is valid and respects adjacency as well as non-adjacency. The final output after the 1st iteration is shown in Figure 2.26. Figure 2.27, shows the output after the 2nd iteration.

For the complete binary tree, $n = 1 + 2 + 4 + 2^h = 2^{h+1} - 1$ i.e. $h = O(logn)$. For a valid labelling of each layer exactly 4 additional elements are required. Total number of layers are $O(\log n)$. Therefore total number of elements required are

$4 * O(\log n)$ which is $O(\log n)$.

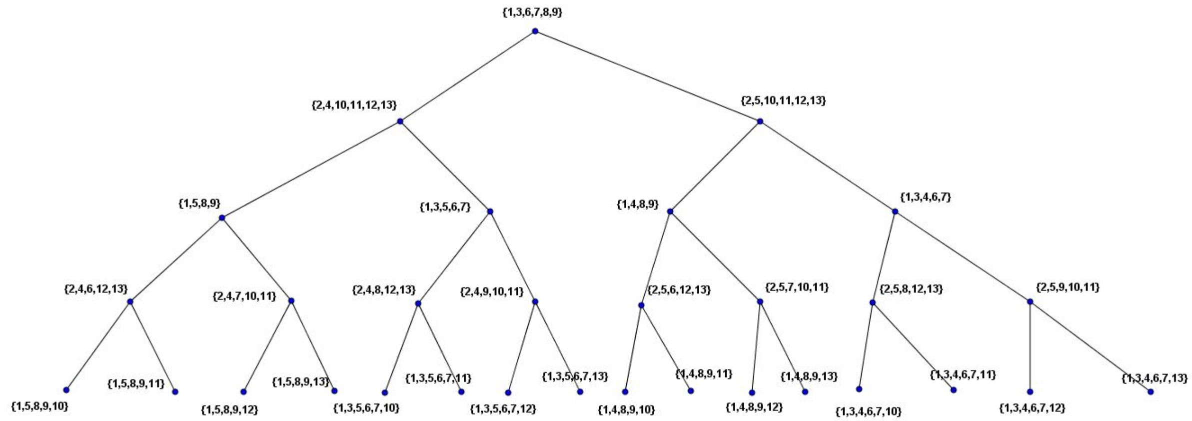


Figure 2.27: After 2nd iteration

Theorem 2.4.2. USN *of any complete k-ary tree is* $O(\log n)$ *where the number of vertices present in the* T *is* n.

Proof. Using a similar algorithm which is described for complete binary tree, it is possible to obtain a valid labelling of each layer of a k-ary tree using exactly k^2 additional elements. k^2 is a constant which is not dependent on n and the total number of layers is $O(\log n)$. Therefore the total number of elements required is $k^2 * O(\log n)$ which is $O(\log n)$. □

It is possible to generate a valid labelling for any tree with the use of following algorithm.

Algorithm 1: Valid-Tree-Labelling

Step 1: For the given tree, calculate its maximum degree a.

Step 2: Consider a complete $(a-1)$-ary tree, with the same number of levels as the given tree with a valid labelling (use Theorem 2.4.2 to obtain a valid labelling).

Step 3: Embed the given tree into this complete tree.

Step 4: For all the vertices which are present in the tree, copy the corresponding labels from the underlying $(a-1)$ ary tree.

The resultant labelling is valid but total number of elements used is not asymptotically minimal.

2.4.1 Discussion on additional features

It is possible to obtain the following additional features by some modifications to our basic labelling algorithm that allow quicker identification of neighbours and non-neighbours. These additional features which are interesting in their own right, prove crucial for some applications of labellings which we present in Chapter 3.

Feature 1: Identification of non-neighbours quickly

Vertices having same label cardinalities are in the same level except for the last 3 levels. So no edge between them.

Feature 2: Identification of neighbours quickly by reducing the search space.

In trees, edges are present only between adjacent layers. After applying the modified labelling algorithm, it is possible to determine the level number correctly of each vertex. This will make searching easier and the user can easily identify object connections.

Explanation of the modified labelling algorithm

The objective of the modified algorithm is to generate labels of unique cardinalities in increasing order for all levels (except the last 3 levels) and assign same cardinality labels to all vertices which belong to the same level. i.e. level 1 must have the minimum cardinality label whereas level i must have labels with the maximum cardinality. In order to obtain the desired features, the following two modifications are required.

1. Input tree. The modified input tree with the valid labelling is shown in Figure 2.28.

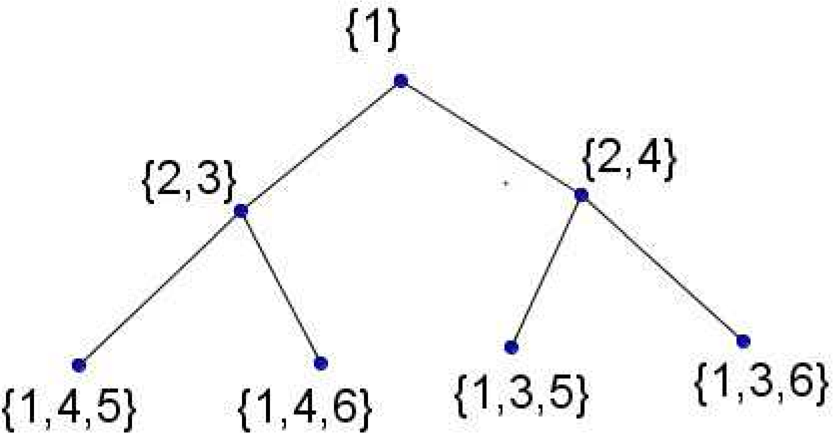


Figure 2.28: Input for the modified algorithm

2. After applying the ValidTreeLabelling algorithm, add {e} to all vertex-labels which are present at level $L + 1$. So overall use of 5 new elements instead of 4.

Let C_i denote cardinalities of all vertex labels which belong to level i. The algorithm adds exactly 1 level after each iteration. The input complete binary tree has 3 levels. Therefore after i iterations, total $i + 3$ levels will be generated. Before applying the modified algorithm, the cardinalities of labels are $1, 2, 3$ (i.e. $C_1 = 1$, $C_2 = 2$ and $C_3 = 3$)
After first iteration, new cardinalities: $5, 2, 5, 4$. After second iteration: $5, 6, 5, 6, 7$. After third iteration: $5, 6, 9, 6, 9, 8$. In general after i^{th} iteration first i entries will be sorted in the increasing order and $C_i = C_{i-2} + 4$ (true for all values of j such that $1 \leq j \leq i$), $C_{i+1} = C_{i-1}$, $C_{i+2} = C_i$ and $C_{i+3} = C_{i+1} + 2$. Using these details, it is possible to quickly identify the location of each vertex in the underlying tree hierarchy.

2.5 Conclusions

The work presented in this chapter has been accepted for the publication. The reference is [45]. The work presented in the Section 2.4 has been published. The references are [31] and [32]. In this chapter, we obtained optimal values of USN for the complement of complete graphs, complete graphs, complete bipartite graphs.

We also obtained asymptotically optimal results for paths, cycles, matching, hypercube, wheel graph, complete binary trees, complete k-ary trees and ladders. These are asymptotically optimal because we obtain O(logn) upper bound which matches the universe lower bound established in Theorem 2.1.1. We also obtained an upper bound on USN on 2-dimensional grid which is square-root of its number of vertices.

In the future we plan to derive optimal and/or lower bound results for hypercubes which is a specially interesting class of graphs because it represents the hasse diagram of the powerset. We would if possible like to obtain an optimal labelling which follows directly from the subsets corresponding to the hasse diagrams. In future, we plan to obtain optimal labelling for harary graph and general bipartitte graph.

CHAPTER 3

Labeled Object Treemap: A New Technique For Visualizing Multiple Hierarchies

In Section 3.1, the relevant key terms are explained. In Section 3.2 key contributions of our work related to treemapping [34] are discussed including:

- Support for data integration in treemaps (social customer relationship management (CRM) tree-map example)
- Key features of Tree-map are discussed briefly including expressive power of tree-map and types of queries supported by it.
- Social network visualization: twitter tree-maps and how they can be used to answer dynamic queries interactively.

Section 3.3 discusses basics of multiple hierarchies including the already known technique for visualization of multiple hierarchies, namely 'Trees in a Treemap'. Other related work is also shown in the same section. Section 3.4 gives a description of our proposed method and its advantages and in Section 3.5, our proposed method is compared with the existing methods with respect to various parameters. Section 3.6 explains details of an implementation interactive version of our proposed method including key features of our implementation. We have developed an interactive version of our proposed method using javascript. The actual code is placed in the Appendix section of the thesis. The final section summarises the results obtained in this chapter.

3.1 Introduction

Key terms which are related to multiple hierarchies are explained in this section.

Hierarchy: A hierarchy refers to an arrangement of items in which items are highlighted as being "below" or "above" or "at the same level as" one another.

Structure: Structure refers to an arrangement and organization of interrelated elements of the system.

Tree structure: Hierarchical nature of the given structure can be represented using tree structure or tree diagram in a graphical form.

Tree: Simulation of tree structure is possible using trees which are widely used data structures for the implementation of abstract data types.

Taxonomy: Taxonomy refers to the practice and science of classification of concepts/things and it also includes the principles that underlie such type of classification. An example of a taxonomy is shown in Figure 3.1.

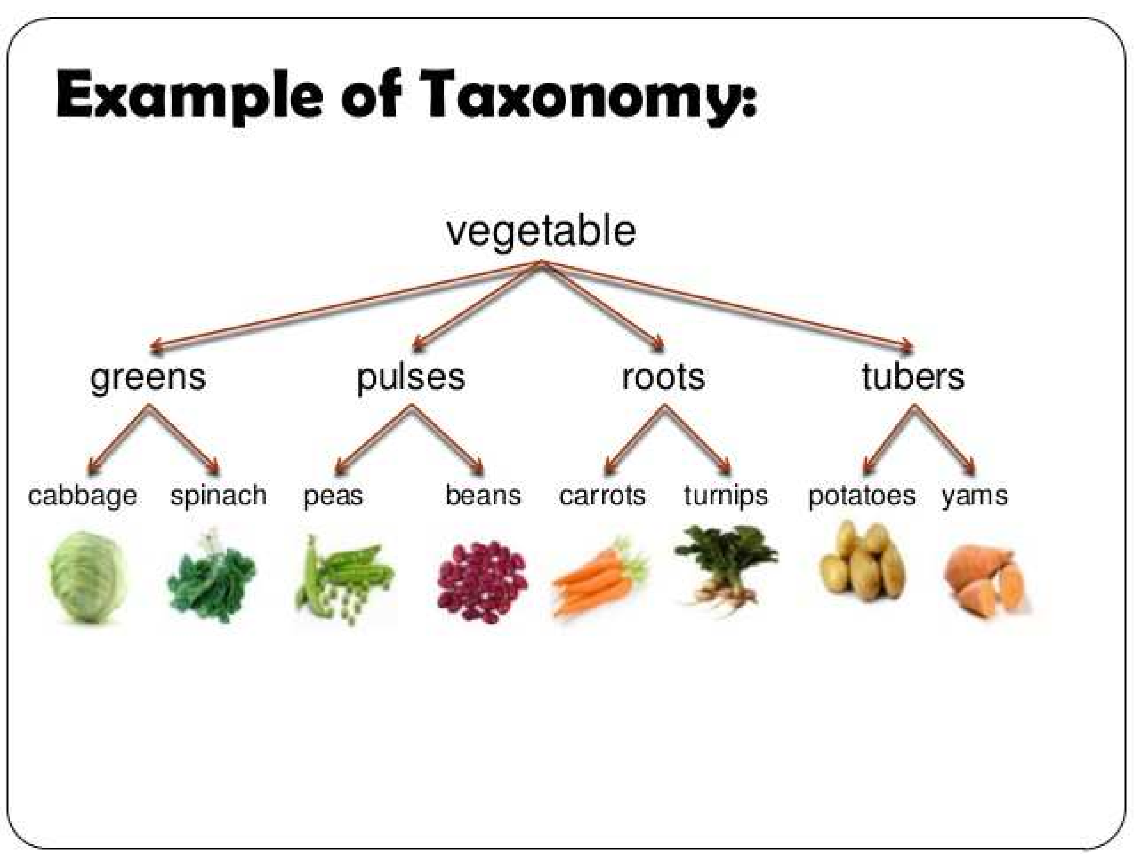


Figure 3.1: Taxonomy

Taxonomy Trees: They are special trees in which objects are represented by leaf nodes only, and classification is represented by all other nodes.

3.1.1 Treemap

The treemap algorithm offers a practical way of displaying large trees (with millions of nodes) in limited space. The approach of treemapping is recursive. One box is allocated for the parent node and children of the node are represented as boxes within it. Practically, using this approach it is possible to render any tree within a standard size display. Treemaps and its variants are studied in detail in the literature.

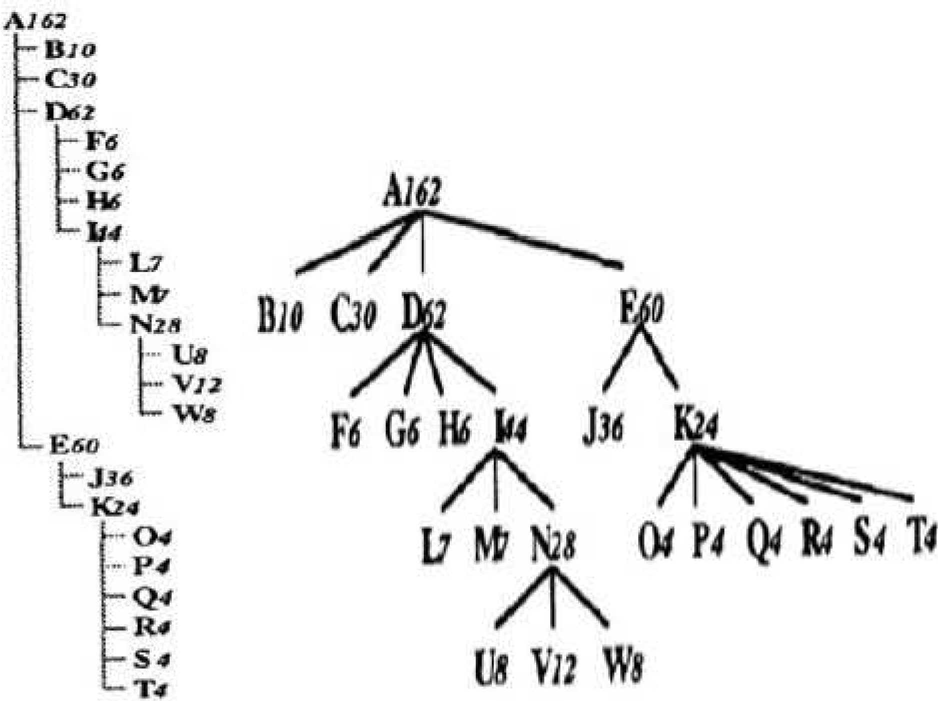


Figure 3.2: Hierarchical data and corresponding tree representation

Tiling is the process of dividing large rectangles into smaller sub-rectangles. Ideally, a treemap tiling algorithm creates rectangles with an aspect ration that is close to one, with ordering based on the underlying data/information tree. Various tiling algorithms are known for tree-maps namely: Binary tree, mixed treemaps, ordered, slice & dice, squarified and strip. Transition from traditional

representation methods to Tree-Maps are shown in Figures 3.2-3.4. In Figure 3.2 some hierarchical data and its equivalent tree representation are shown. One can consider nodes as sets, children of nodes as subsets and therefore it is fairly easy to convert a tree diagram into Venn diagram. Figure 3.3 represents a Venn diagram and its equivalent representation as a nested tree-map. A nested tree-map represents the nesting of rectangles. Finally in Figure 3.4, tree-map representation of a given hierarchical data is shown. Tree-map is a comprehensive design in which a border is used to show nesting and it is more space efficient compared to the nested version. Key advantages of tree-maps are easy identification of patterns and efficient usage of space.

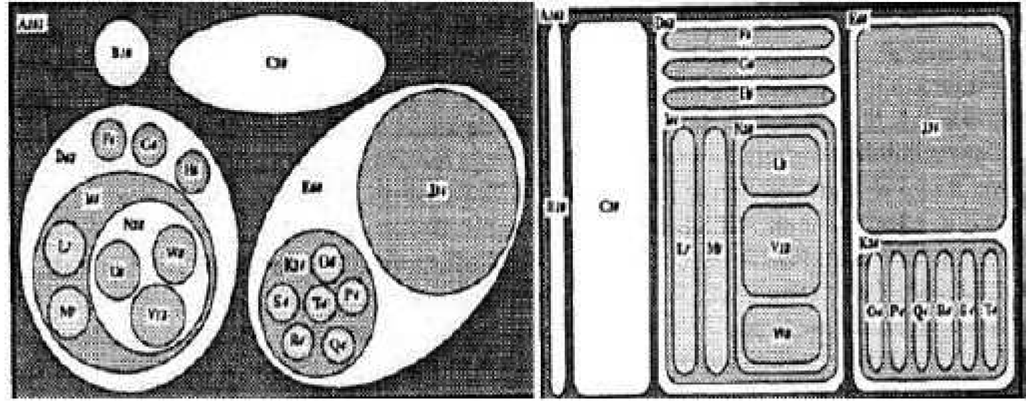


Figure 3.3: Venn diagram and nested tree-Map

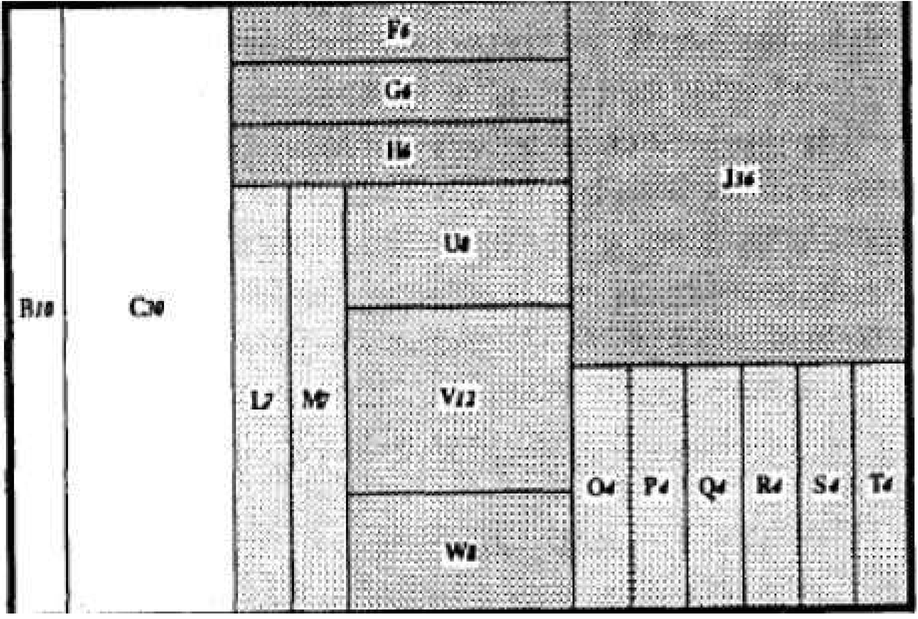


Figure 3.4: Tree-Map

Queries can be answered easily with the use of tree-map visualization. Consider tree-map representation of an operating system, say UNIX. With the help of this representation, one can easily answer following queries: Identification of the directory which is taking up the most space, how much space is taken up by specific directories, types of files present in hierarchy etc. Tree-maps offer dynamic visualization. Key features of dynamic visualization are: immediate feedback mechanism, support for dynamic queries (queries which are incremental and reversible).

Figure 3.5: www.peets.com

Intuitively, tree-map representation is better than simple manual list represen-

tation. Peet is a San Francisco Bay Area based famous coffee roaster as well as retailer since 1966. A marketing survey showed following result: For 92 out of 100 customers of peet, (who used the tree-map interface) online shopping was easy. Whereas for the manual lists users, this percentage was only 12. Tree-map interface of peet is shown in Figure 3.5. Most of the other techniques of data visualization were invented in the absence of widely-available computational (computer) resources. Tree-maps were conceived as a result of computerization and therefore they have crucial benefits from this more modern scenario.

3.2 Tree-map: a visualization tool for large data

Tree-Maps are used to present hierarchical information on 2-D [58] (or 3-D [11]) displays. Tree-maps offer many features: based upon attribute values users can specify various categories, users can visualize as well as manipulate categorized information.

The traditional approach to represent hierarchical data is to use a directed tree. But it is impractical to display large (in terms of size as well complexity) trees in limited amount of space. In order to render large trees consisting of millions of nodes efficiently, the Tree-Map algorithm was developed. Even the file system of UNIX can be represented using Tree-Map. The definition of Tree-Maps is recursive: allocate one box for a parent node and the children of that node are drawn as boxes within it. Practically, it is possible to render any tree within a predefined space using this technique. It has applications in many fields including bio-informatics, visualization of stock portfolio.

3.2.1 Guidelines for tree map design

1. Every box of the tree-map can display two different measures namely size and color. Size should reflect quantity measure whereas color is used to display measure of performance and/or change. i.e. satisfaction of customer, growth rate etc.

2. In selection of tree-map layouts, extreme aspect ratios should be avoided [38].

3. Tree-maps are more suitable for high density data; for low density one can use bar charts.

4. Comparing non-leaf nodes is easier in tree-maps compared to bar charts.

5. Appropriate labels should be given and labels should be meaningful.

6. It is advisable to show labels only when the user rolls over a tree-map box.

7. Labels must be visible in the multicolored background of tree-map.

8. Depending on the nature of the color measure, one sided/two sided color range should be used.

9. In order to show correlation, highlighting should be used.

10. One can use animation in tree-maps to show change in the data.

11. Simple presentation method (Tooltip window/sidebar) can be used to show node detail.

3.2.2 Expressive power of tree-map

Tree-Maps are used to express a variety of nested as well as hierarchical data and data structures. In general, the type of tree-map representation to be used should depend on the application and the type of data hierarchy.

"Tree-map visualization generators" are used to display tree-maps for arbitrary hierarchical data. Tree-Maps can be provided as images in static form or they can be used to provide interactive features (like zooming into a small area of the hierarchy) in applications. Tree-maps support browser as well as rich client applications. In one of the applications, tree-maps are incorporated with Windows Forms- Microsoft Corporation.

Tree-Maps are also famous amongst news designers. Examples are listed below.

1. NewsMap [48] (Newsmap.jp is developed by Marcos Weskamp and it represents current items of Google News using an interactive Tree-map, which is shown in Figure 3.6.)

2. London 2012 Olympics and Tree- maps [16].

3. BBC News- SuperPower: Visualising the internet

4. The New York Times- Obama's budget proposal (Year 2011)

5. CNN Twitter buzz of South Africa (Year 2010)

Figure 3.6: www.newsmap.jp

3.2.3 Social network data and tree-map

For the promotion of brand, the role of a marketer is not significant in the modern era of social media. In the past, information was produced by marketers and consumed by customers. Currently more information is generated by customers about brands on social media including blogs, social media networks, online forums. Currently, marketing teams are struggling with analysis of this online information, which is required for prediction of acceptance rates of products, patterns

of purchase and levels of satisfaction in customers. Marketers can use these new channels for promotion by developing customers as brand advocates.

For travel as well as hospitality industry, decisions related to purchase are mainly determined by online reviews as well as recommendations. Online customer data along with business functions information forms an integrated database. In order to study levels of customer loyalty, study of this integrated database is necessary.

It is possible to use customer tree-map for segmenting customers and generation of 'brand score' for customers. Brand score depends on 1) Brand engagement of customer (behavioral aspect) and 2) Attitudes of customers.

Two different types of score namely spend value score and advocacy score are calculated using integrated database (traditional CRM and unstructured data). Social CRM tree-map can be created by plotting these scores (by integrating two data-sets) on a 2-D axis. An example of social CRM tree-map is given in Figure 3.7. Members without any spend value are defined as noncustomers. This tree-map is useful for calculation of overall "customer brand score".

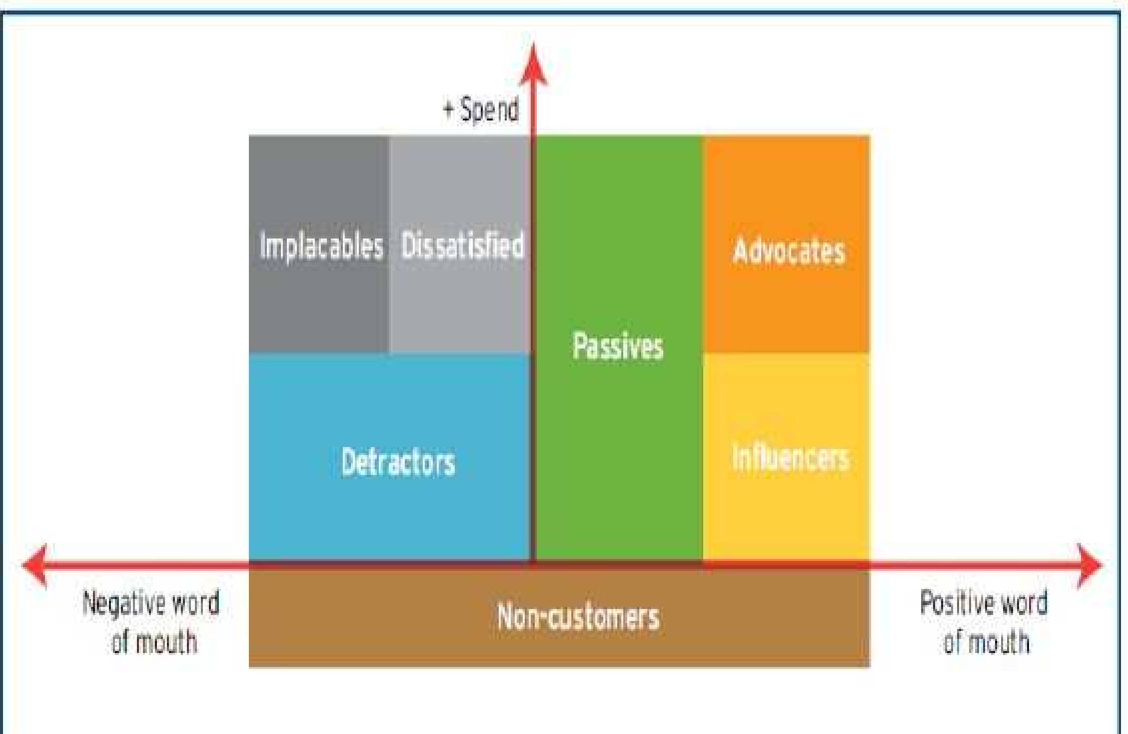


Figure 3.7: Social CRM tree-Map

Advocates have the following qualities: They have high values for spend value as advocacy score. They are brand evangelists and their behavior as well as attitude is very loyal to a brand.

After successful development of Tree-map, organizations can take actions in order to cultivate advocates of brands.

3.2.4 Types of queries supported by tree-maps

Tree-Maps provide two important features by supporting dynamic queries:

1. Querying a large set of data.
2. To find patterns in large data sets [56].

In tree-maps, dynamic queries are implemented using radio buttons, buttons and sliders. Tree-map follows the principle of direct manipulation for searching a large database.
Key features of query processing of Tree-Map are listed below:

- Supports visual representation (for components of query).
- Supports visual representation of query results.
- Provides rapid, reversible and incremental control of query.
- Selection is done by just pointing, not by typing.
- Tree-map provides immediate as well as continuous mechanism of feedback.

3.2.5 Tree-map for Twitter data visualization

Key requirements for visualization of any social network are listed below:

- Identification of the actors -members of the social network.
- Visualization should represent relationships of various types.

- Visualization should support aggregated as well as structured views of the complex social network.

Consider the example of Twitter network with four sample actors namely Steve, John, Luke and Adam. Figure 3.8 represents this network as a Tree-map. Tree-map offers all the crucial features which are desirable for a visualization tool. Here actors are represented by rectangles and the size of each rectangle is proportional to the total number of tweets sent by that particular actor. The friendship relationship is represented by a common edge between two rectangles. In our example, the rectangle corresponding to Luke has highest area which implies the highest number of tweets amongst the four users. No common edge is present between Steve and Luke which implies that they are not friends on Twitter.

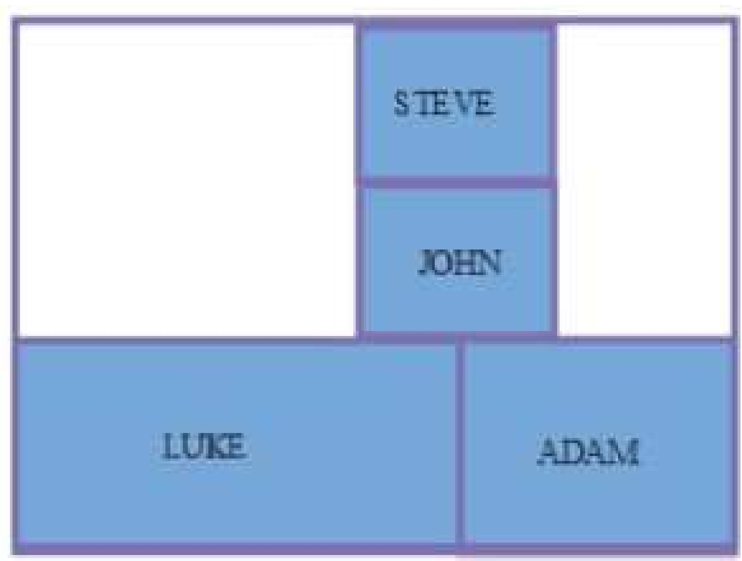


Figure 3.8: Basic Twitter treemap

Other variants of Twitter tree-map are also shown in Figure 3.9 and 3.10. Tree mapping is not as popular as other visualization techniques, still recent survey results are encouraging for twitter tree-maps [55]. Better results are possible by improving current design of tree-maps as well as integration of tree-map with other visualization techniques.

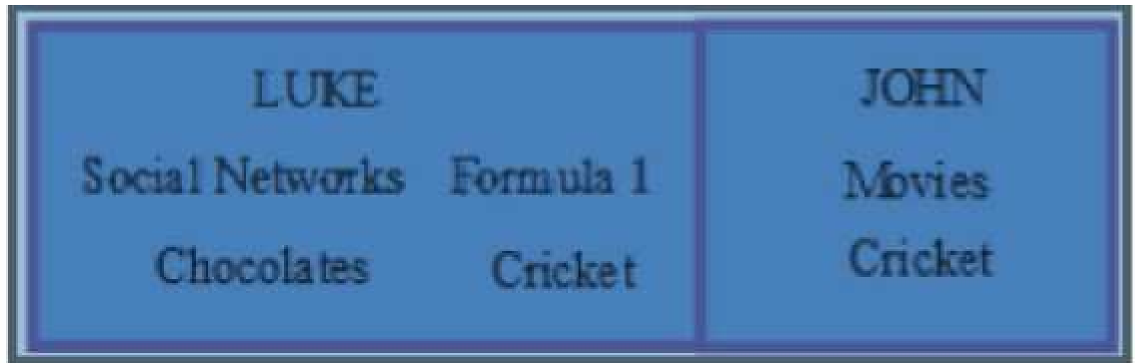


Figure 3.9: Twitter treemap with additional information (actor's interests)

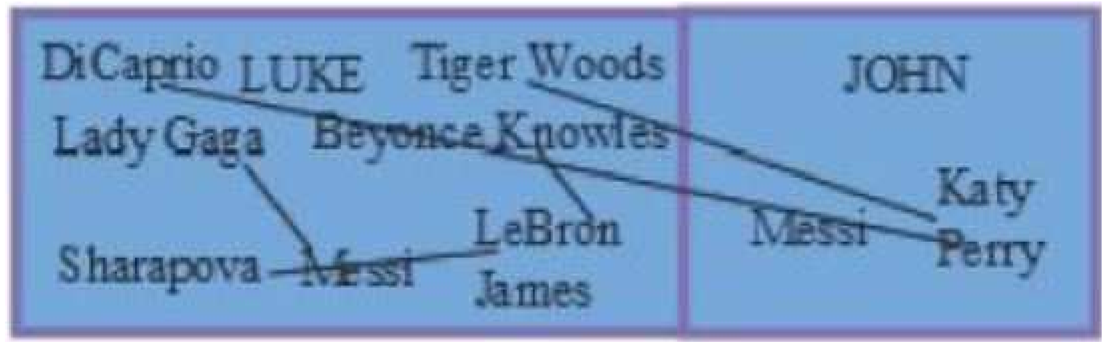


Figure 3.10: Twitter treemap integrated with network diagram

3.2.6 Discussion on interactivity of tree-maps

Tree-maps offers interactive features which are distinctive. The main objective of this visualization tool is to provide an interactive display on a computer screen. Because of this unique feature, one can explore the data hierarchy effortlessly and simultaneously decent level of estimation is also possible for quantitative aspects of the information. In order to provide element specific information in detail, various tree-map software offer computer screen mouseovers using which the user can get specific information just by placing the computer mouse over the specific box. Because of these crucial interactive features, tree-map is an emerging powerful visualization technique; for large social data-sets because real time feedback is essential in the case of a complex social network. The analyst can use this interactive feature to traverse the tree and can also present categorical data view at every level.

Generally, queries on social network data focus more on relationships between

different groups and sizes of particular categories is a very common type of query. For example, which country has the highest number of twitter users? Now consider the following complex query: Do white males in North America use twitter more than white females in South America? In order to answer this question one has to consider sub-questions for all data points. i.e. whether a particular person is black/white and has a twitter account or not and so on.

In order to answer these queries interactively for categorized social data, we propose the use of CatTrees (an enhancement of tree-maps) [37]. It is possible to answer these types of question easily if the data has hierarchy because then, for each possible answer pattern, one can allocate a leaf node with counter and to get the final answer, the analyst can follow two different paths (depending upon query) from root to leaf nodes and give the final result depending upon the comparison of counters. So depending upon query, a new hierarchy may be required every time. In short, dynamic hierarchies are required to support dynamic queries! Dynamic hierarchies are implemented by CatTrees.

All social data is not hierarchical in nature. Surprisingly tree-maps can be used to visualize non-hierarchical data too. In this case, an imaginary hierarchy is provided as an input by the analyst [59].

3.3 Visualizing multiple hierarchies

Multiple Hierarchies: A hierarchy refers to an arrangement of items in which items are highlighted as being "below" or "above" or "at the same level as" one another. In trees, edges are present between adjacent levels (just above and just below the given level) only and no edges are present between nodes which are present at the same level. Therefore, trees are an appropriate data structure to represent hierarchies and for any two nodes (objects) of a tree, it is easy to establish relationships including being 'below' or 'above' or 'at the same level'. Therefore visualization of multiple hierarchies is equivalent to visualization of two distinct trees namely 1) taxonomy tree 2) the other object tree where object connections are shown and each object is related to the leaf nodes (objects) of a taxonomy tree.

The simplest solution is shown in Figure 3.11 in which both the trees are drawn side by side. Here, it is not possible to identify frequencies as well as locations of different object types quickly in the underlying object tree.

For many applications it is necessary to consider two aspects: the relationship between different objects and identification of the object type. One can show these aspects using two different trees: 1) taxonomy tree 2) the other tree where each node is related to leaf nodes (objects) of a taxonomy. The problem is to design a visualization technique which effectively conveys both the desirable features i.e. relationships between different objects (object tree) along with the mapping of each object with taxonomy.

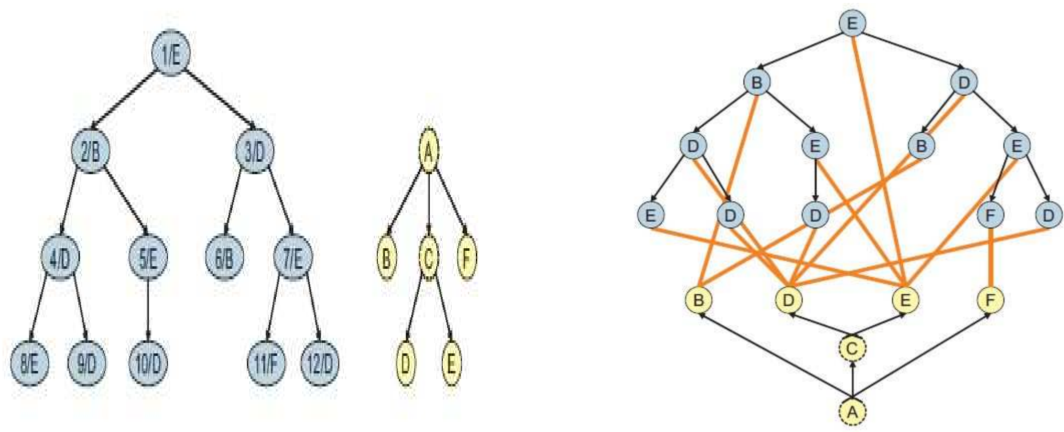


Figure 3.11: a) Two separate tree diagrams and b) Linked tree diagram

To the best of our knowledge, the best-known visualization technique for representing multiple hierarchies is: Trees in a Treemap technique [12]. The simplest solutions are shown in Figure 3.11 in which a) both the trees are drawn side by side. Here, it is not possible to identify frequencies as well as locations of different object types quickly in the underlying object tree. b) connections between the two trees are shown using additional edges. Neither of these solutions is efficient for big data.

3.3.1 Related work

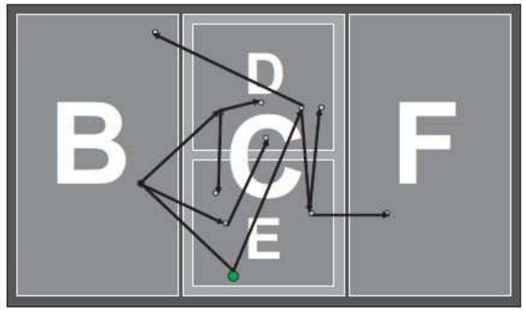
Trees in a treemap technique belongs to the Agglomeration class of visualising multiple trees [23]. Agglomeration: Here a single representation is used in order to display multiple parents of a node. One possible option is to replicate nodes with at least 2 parent links across the trees in order to maintain similarity of the overall structure to a tree. Node link representations are also used in which multiple tree structure is shown as a graph structure (structure with cycles). For example, GLAD system [17]. Directed placement of various overlapping treemaps is used in CristalView [46] in which shared nodes are used to communicate across multiple hierarchies. The success of the selected visualization technique depends upon various factors including target audience and user interface [2]. Other related visualizations are ZTree [6] , Multitrees [19] and animation [53].

3.4 Description of our proposed method and its advantages

Labeled object treemap technique is very similar to trees in a treemap technique. In the trees in a treemap technique, nodes are represented by small circles which are placed in the boxes that represent the objects associated with the nodes. Edges are represented by lines which connect these small circles.

According to us, Trees in a treemap method (see Figure 3.12) has the following problems:

1. Edge crossing is present in both the variants of the technique.
2. Continuity: For very large data sets, it is very difficult to follow the edge in order to identify whether those two objects are connected or not.



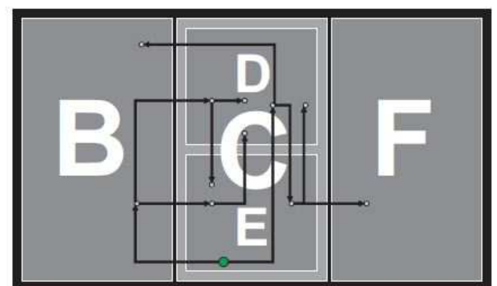


Figure 3.12: Trees in a treemap visualization technique (from [12])

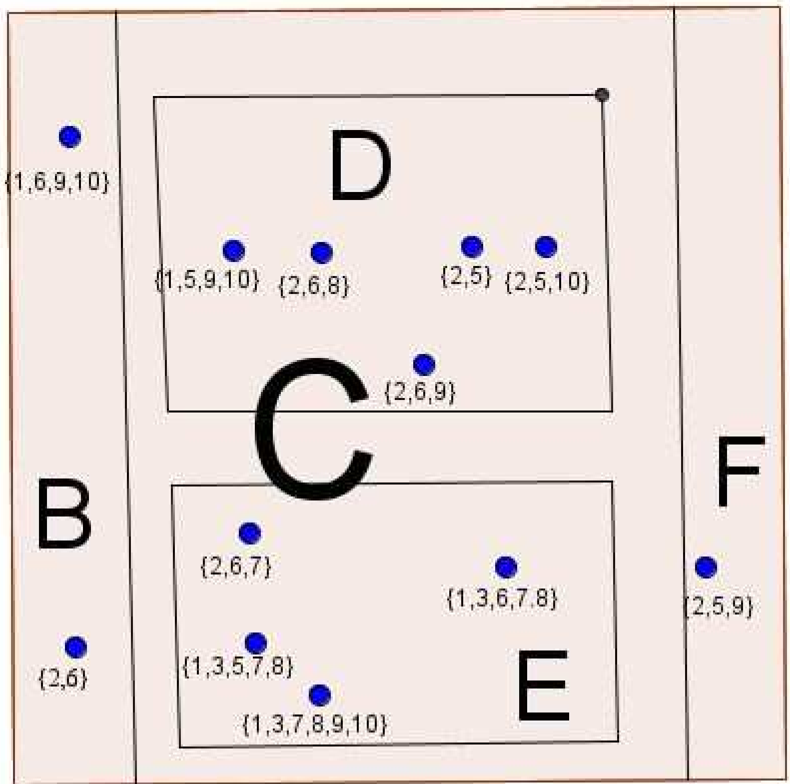


Figure 3.13: Our proposed method: Labeled Object Treemap

Our proposed method solves these problems by constructing an equivalent labeled representation which does not contain edges and in the case of complete k-ary tree, the total number of labels which are used in the representation are asymptotically minimal. i.e. $O(\log n)$ (see Theorem 2.4.2).
If the labels of two objects are disjoint then they are adjacent. i.e. edge is present between them.

Visualizing a complete k-ary object tree with the given taxonomy:

The size of each label is at most $O(\log n)$. Hence, it is possible to represent the information about edges using an $O(n) * O(\log n)$ matrix. Each row in the matrix corresponds to the label of a node. If element j is present in the label of the ith node then entry $(i, j) = 1$, else 0. In the worst case $O(\log n)$ comparisons are required in order to determine whether two nodes are adjacent or not. In the best case, only constant time is required for the same query because it may possible that the 1st element is present in both the labels.

Advantages: It offers a single representation in order to visualize object tree as well as taxonomy. Crossings are not present in the representation. Continuity is not visible but one can still obtain the required information by a careful observation of labels. It is possible to detect clusters and outliers using our proposed method. The visualization technique is highly compact because it requires just $O(n) * O(logn)$ space (for complete k-ary object trees) in order to store information about edges. Taxonomy is represented using treemap.

From Chapter 2 (see Valid-Tree-Labelling algorithm), it is clear that it is possible to obtain a valid labelling of any given tree. The final visualization is shown in the Figure 3.13.

3.5 Comparison with existing methods

Calculation of leaf word of the taxonomy:

$lw(t) = t$ if t is a leaf node

$=$ concatenation of $lw(t_1), lw(t_2), \ldots, lw(t_k)$ otherwise

Here $t_1, \ldots, t_k$ are subtrees of t.

For the given taxonomy, valid leaf words are $BDEF$, $FEDB$ and $DEBF$.

In the space filling visulization technique, adjacency matrices can be used. In Figure 3.14, adjacency matrix representations are shown for sorted as well as unsorted dimensions. For sorting, a leaf word is used.

In colored tree diagrams, colors are assigned to leaf nodes. This assignment is based on the order of occurance of a leaf node in the leaf word. Similar colors

are assigned to objects which are close to one another in the underlying taxonomy (see Figure 3.15). Leaf word is also used for sorting objects in Parallel coordinate views technique (see Figure 3.15).

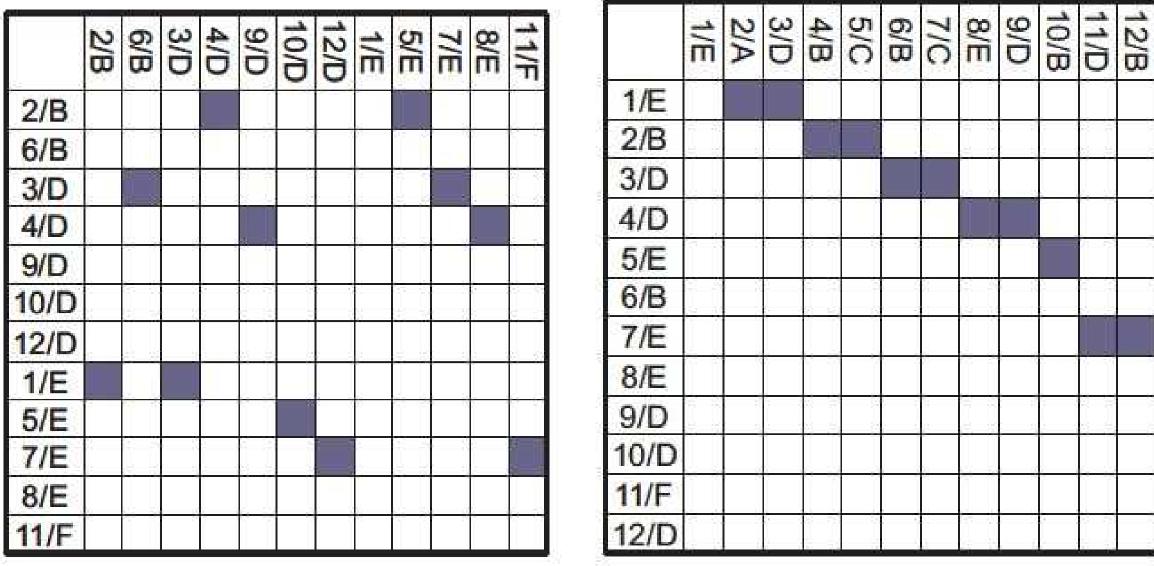


Figure 3.14: Adjacency matrices (from [12])

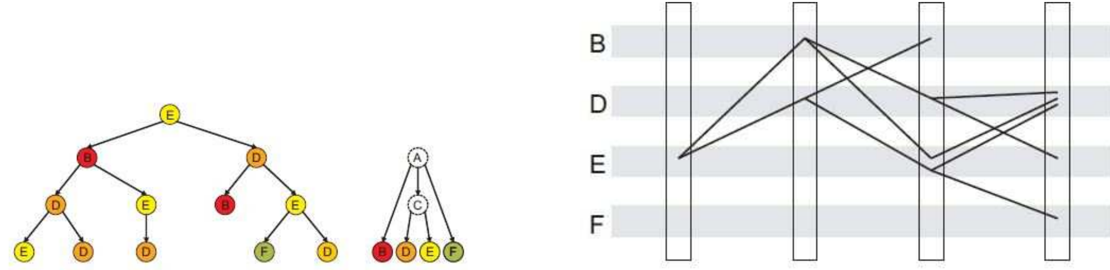


Figure 3.15: Colored tree diagrams and Sorted parallel coordinates (from [12])

We are going to consider following criteria for different visualization techniques (for multiple hierarchies) in order to compare them with our proposed technique.

Single representation: We are supposed to display relationships between two distinct structures: object tree and taxonomy. Therefore it is desirable to have a single representation for the provided multiple hierarchies.

Crossing: There is no restriction on the size of the data (object tree). So edge crossing in the drawing must be reduced in order to see the connections between

objects more clearly.

Continuity: This parameter reflects the difficulty level in following the lines which represent edges in the underlying visualization technique.

Compactness: The visualization technique should be as compact as possible.

Table 3.1: Comparison with existing techniques

Visualization Technique	Single Model	Crossing	Continuity	Clusters Outliers	Compact	Taxonomy
Separate Tree Diagrams	No	No	Straight	No	No	Tree Diagram
Linked Tree Diagrams	No	Yes	Straight	Difficult	No	Tree Diagram
Colored Tree Diagrams	No	No	Straight	Yes	Medium	Color
Unsorted Matrix	Yes	No	Not Visible	No	High	Not Visible
Sorted Matrix	Yes	No	Not Visible	Yes	High	Order
Sorted Parallel Coordinates	Yes	Yes	Straight	Yes	Medium	Order
Trees in TM (straight lines)	Yes	Yes	Straight	Yes	High	Treemap
Trees in TM (orthogonal lines)	Yes	Reduced	Orthogonal	Yes	High	Treemap
Labled Object Treemap (our proposed method)	Yes	No	Not visible	Yes	High	Treemap

In order to compare different techniques, representation of the same given trees (shown in Figure 3.11) are displayed using various methods.

3.6 Interactive labeled object treemap

We have designed an interactive version of this technique. In our interactive solution, for the given set of object types, the user can provide any size of input tree in the given textbox. A unique set label is generated for each node and nodes are distributed and displayed (as small red circles) according to their object types in the rectangle areas of the treemap. Four check-boxes are provided to offer various

features. With **internal edges** and **external edges** options selected, whenever a user clicks on a node, all the neighbors of that particular node (including the node itself) will be highlighted in black colour along with edge connections and IDs. For the object of interest, users can also exclusively identify its neighbors of similar object type or different object type(s) by selecting internal or external edges check-box respectively and clicking on the object. **IDs only** option is important to identify object names and hence object of special interest and its location in the visualization so that the user can explore it more by selecting an appropriate option and clicking on it. **All edges** option shows all the connections (edges). Our proposed interactive visualization technique has the following desirable features: single representation, high compactness, cluster identification and solves the issues of edge crossings and continuity.

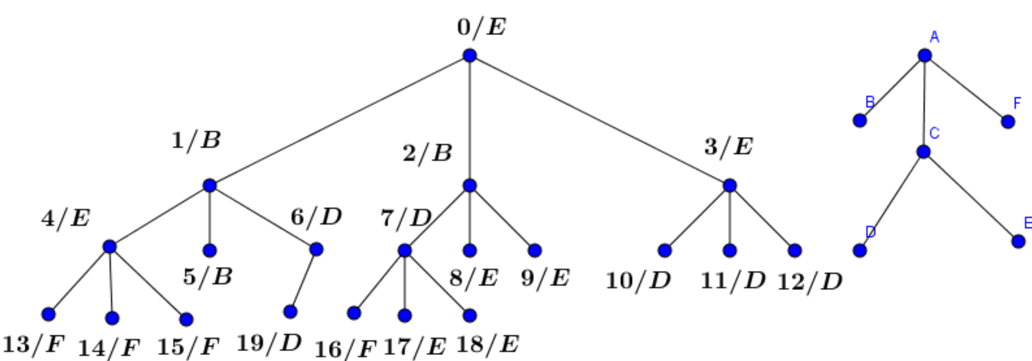


Figure 3.16: Input for our proposed technique: object tree and taxonomy tree

In order to make the existing visutalization technique user-independent and interactive, we have implemented a dynamic interactive version of our proposed visualization technique 'Labeled Object Treemap'. The same taxonomy tree (which is given in Figure 3.16) is considered for our implementation. Treemap is used in the background in order to highlight taxonomy. Object classes having same parents in the taxonomy tree are highlighted using the same colour. For the given input (shown in Fig. 3.16), the overall layout of our proposed system is, as shown in Fig. 3.17.

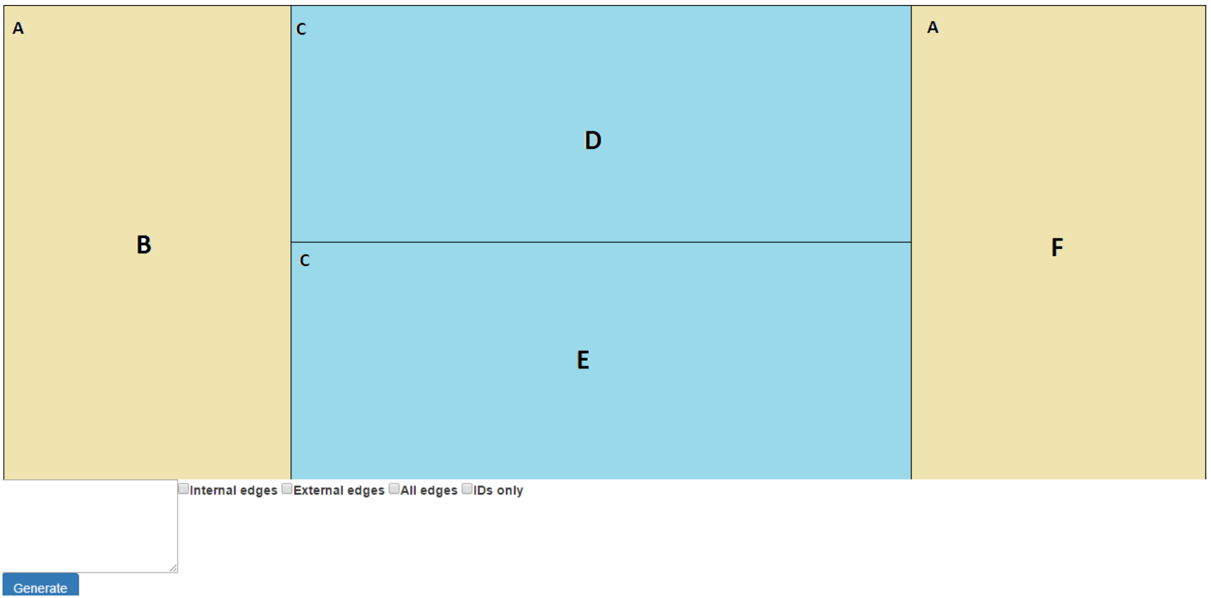


Figure 3.17: The overall layout of our proposed interactive visualization

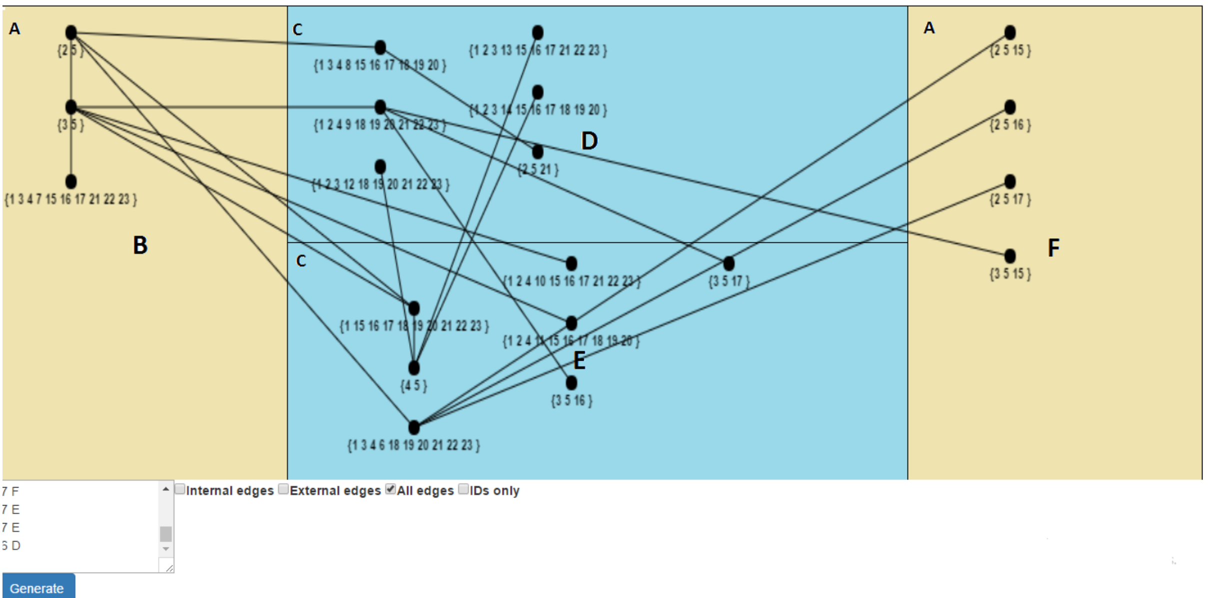


Figure 3.18: All edges option

3.6.1 Key features of our implementation

Visualization for any given input tree: The visualization technique uses label construction methods explained in the Section 2.4. It is possible to identify object types and object relations using our proposed work. Users are expected to provide input in the textbox in (Parent node ID, Object type) format. For example, $(2, C)$ refers to the node with object type C and parent node 2. Here, the parent

ID is with respect to the object relationship tree. The root node doesn't have any parent so for the root node, we use -1 as a parent node ID. After giving a valid input tree, the **Generate** button should be clicked and the user can see the visualization where nodes are positioned depending upon their object types along with their unique labels. Objects are connected if and only if their corresponding labels are disjoint.

Identification of object names: By clicking on the IDs only button (see Fig. 3.19), labels will be replaced by IDs (object names). Using this feature, a user can identify specific object(s) in which he/she is interested.

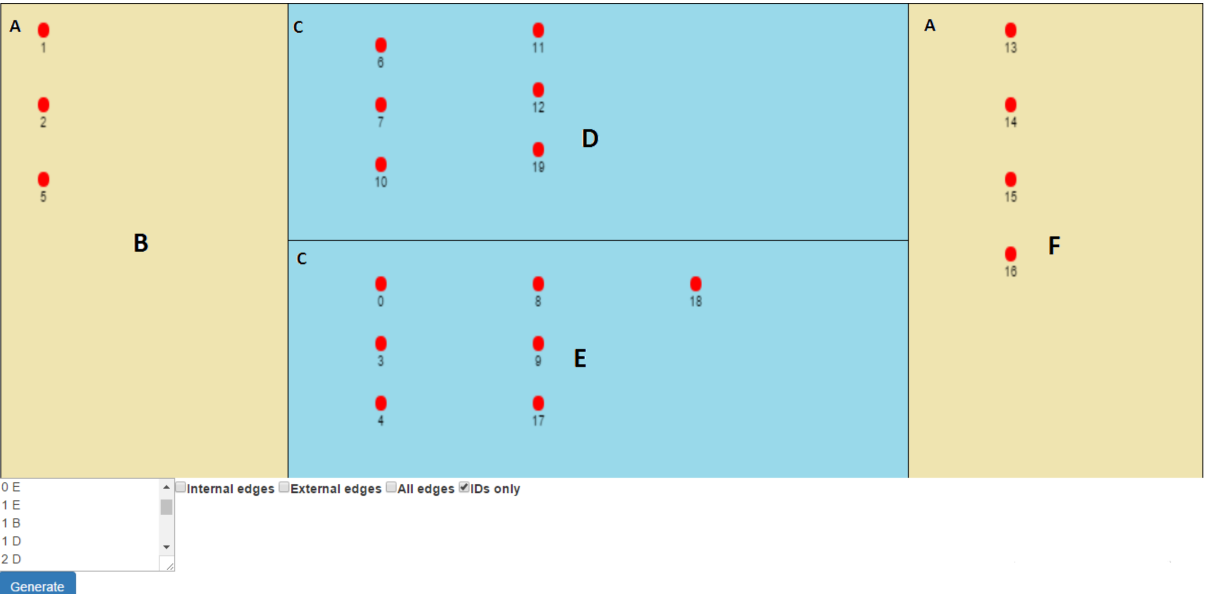


Figure 3.19: IDs only option

Visualizing connections: Two different checkboxes namely Internal edges and External edges are provided to identify neighbours of a node. In the Internal edges option (see Fig 3.20): After clicking on any particular node (say v), v is highlighted in black along with its ID. All the neighbours of the node v which belong to the same object type are also highlighted (with their IDs) in black colour along with black edges to show connections. Colours of all other nodes (which are non-adjacent to v/nodes having different object types than v) will remain red. In the External edges option (see Fig 3.21): After clicking on any particular node (say w), w is highlighted in black along with its ID. All the neighbours of node w which belong to other object types are also highlighted (with their IDs) in black colour

along with black edges to show connections. Colours of all other nodes (which are non-adjacent to w /nodes having the same object type as w) will remain red. In order to see all the neighbors of a particular node, both of the above mentioned options must be selected before clicking on any particular object (see Fig. 3.22). It is also possible to visualize the whole tree by selecting 'All edges' option (see Fig. 3.18).

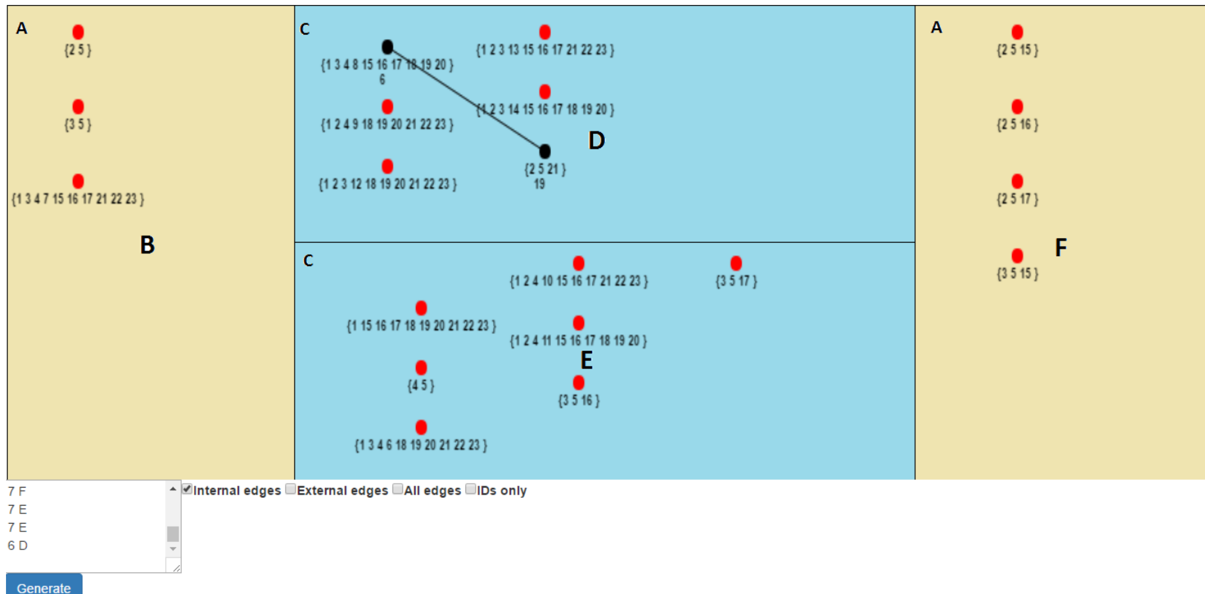


Figure 3.20: Visualizing neighbours of similar object type for the object node 10

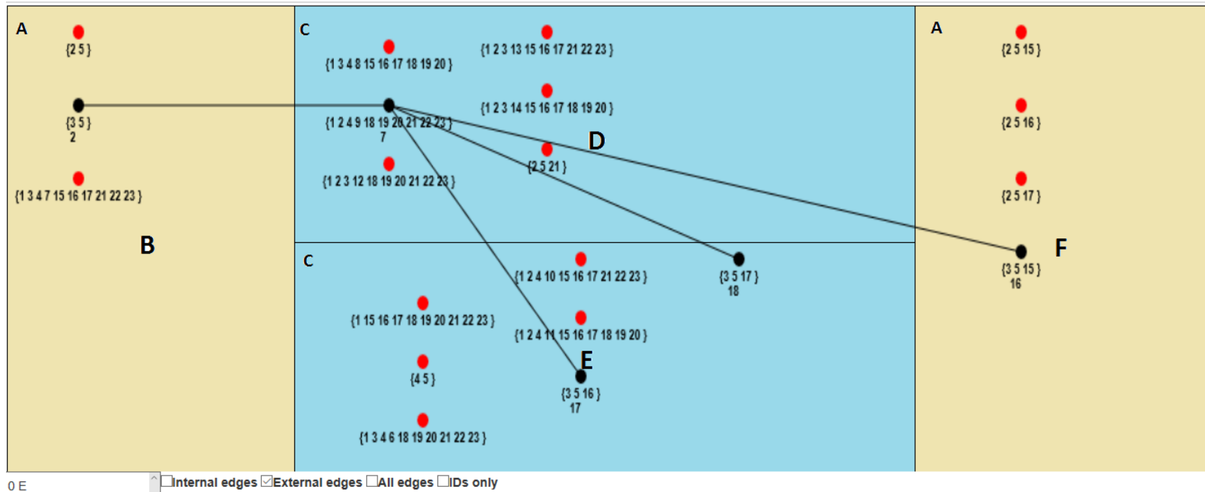


Figure 3.21: Visualizing neighbours of different object types for the object node 7

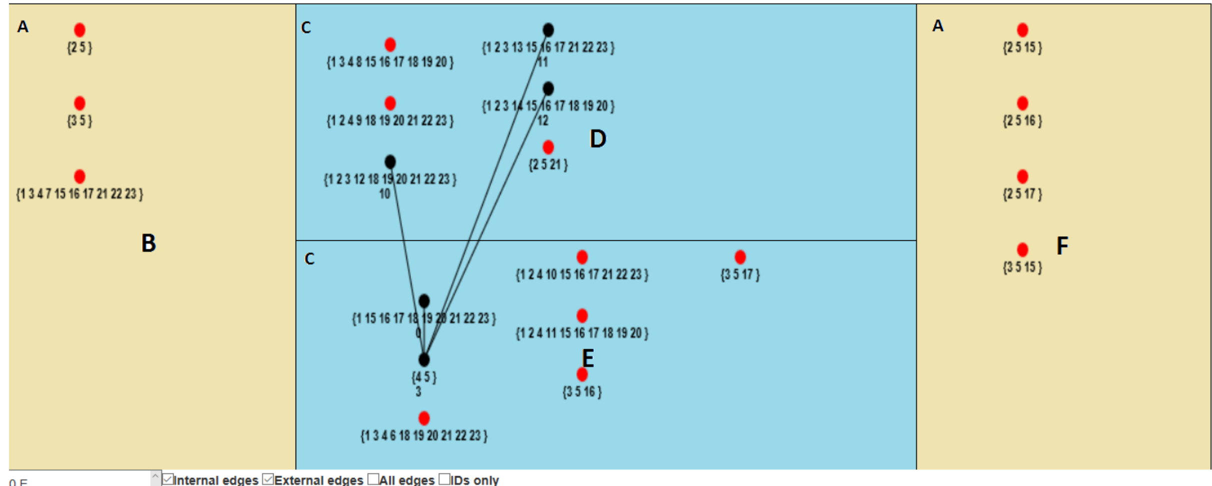


Figure 3.22: Visualizing all the neighbours of object node 3

3.7 Conclusions and future work

The work presented in this chapter has been published by us. The references are [34] [32] [31].

Our proposed data visualization technique shows all the data (multiple hierarchies) without any information loss. Using our technique, it is possible to display large data sets coherently. The method encourages the human eye to compare different objects because each object is represented by a unique label. It is possible to identify objects which obey certain rules with the use of the underlying treemap structure. As shown in Table 3.1, our proposed method offers all the good characteristics of existing methods including single representation, identification of clusters, compactness as well as visible taxonomy using treemap.

The interactive technique of our proposed technique generates labelling of all objects automatically. It also offers various features using which it is easy to identify specific types of neighbors of a node just by clicking on it. Taxonomy is visible using treemap.

In future, one can think of an arrangement of nodes that is more compact so that a larger number of nodes can be clearly shown (along with the labels) in the same available space. Perhaps, arranging the nodes on a zigzag pattern is one such solution.

In future, to improve efficiency in computation, we plan to assign an extra label to each node which represents the level number. Edges are present only between adjacent layers so this idea may reduce the search space. Within one rectangle region of treeemap, we will put disjoint labels i.e. adjacent vertices within the alpha-neighborhood of them. This will help users to identify connections rapidly. To provide more information related to adjacency among the nodes, we will align adjacent nodes across different rectangles of treemaps in the same horizontal/vertical line. In order to provide more information about the underlying hierarchical structure, we may plan to use other variants of treemap: ordered/cushion treemap.

CHAPTER 4

Edgeless Graph: A New Graph Based Information Visualization Technique

In Section 4.1, we list some applications of graph visualization and present some of the challenges involved. Effects of dynamic changes (adding and/or removal of vertices and/or edges) on USN are discussed in Section 4.2. The section explains an algorithm using which it is possible to generate a valid labelling of any given input graph. A brief description of our proposed methods is given in Section 4.3. Section 4.4 explains how our proposed technique can be used for the analysis of a social network and dynamic changes in the underlying social network are also considered. The final section summaries the work done in this chapter.

4.1 Introduction

Applications of Graph Visualization are in many areas. Some of them are listed below:

- Representation of hierarchical structures using trees
- Website maps
- History of internet browsing data
- Biology and chemistry (for the representation of phylogenetic trees, genetic maps, molecular maps etc)
- Other applications include data flow diagrams, entity relationship diagrams, logic programming.

Major challenges in Graph Visualization :

Major challenges in graph visualization are related to graph drawing. It is difficult to display a dense graph (with a large number of vertices and edges) within the available 2-D display interface. It is necessary to reduce edge crossings [49] and avoid long edges with bends. If the graph is very dense, it is even hard to distinguish vertices from edges. A good representation technique should have effective interactive solutions for the above mentioned challenges.

4.2 Effects of dynamic changes (adding and/or removal of vertices and/or edges) on USN

Results related to effects of dynamic changes (adding and/or removal of vertices and/or edges) on USN are given below.

We first deal with addition of a vertex.

Theorem 4.2.1. *$G' = G + v$ where order of G is n.*
USN(G)$\leq$ USN$(G') \leq$ USN(G)$+(n-1)$

Proof. Consider a graph $G(n, m)$ with USN k. After adding a new vertex v and m' edges between v and some m' distinct vertices of G, consider $G'(n+1, m+m')$.
Observation: USN(G') is at least k.
This can be rephrased as a lower bound on increment in USN, upon adding a vertex is 0.
Proof by contradiction:
Suppose USN$(G') \leq k-1$
Now after removing vertex v (and all edges incident on it), the remaining graph= G has a valid labelling with USN$(G') = k-1$ which is not possible because optimal labelling has USN$(G) = k$. (Initial assumption)
Upper bound on increment: $n - d(v)$
Idea:
A label of v should have some elements from the labels of all the vertices which are non-adjacent to it. Number of non-neighbours are $n - d(v)$ for v and hence in the worst case $n - d(v)$ new elements are required.

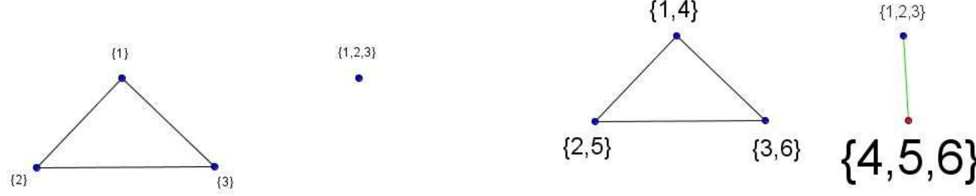


Figure 4.1: Increase(decrease) in USN after adding(removing) a vertex

The tightness of the upper bound result is shown using following example. In the worst case, the existing graph G may have clique of size $n-1$ and one isolated vertex, say v_i with $|l(v_i)| =$USN. Consider the case where the newly added vertex v is adjacent to only v_i. v is non-adjacent to the clique of size $n-1$ and in order to preserve non-adjacency with the clique, $n-1$ extra elements are required. $\square$

The next Theorem is essentially a rewording of previous result as deletion of a vertex is a reversal of addition of a vertex. The minor results also follows.

Theorem 4.2.2. *$G' = G - v$ where order of G is n.*
USN$(G) - (n-1) \leq$ USN$(G') \leq$ USN(G)

Theorem 4.2.3. *$G' = G \setminus e$,*
USN$(G) - \min(x,y) \leq usn(G') \leq usn(G) + 1$, *where* $e = (v_a, v_b) \in E(G)$ *and* $x = |l(v_a)|$ *and* $y = |l(v_b)|$

Theorem 4.2.4. *$G' = G + e$,*
USN$(G) - 1 \leq usn(G') \leq$ USN$(G) + \min(x,y)$, *where* $e = (v_a, v_b) \notin E(G)$ *and* $x = |l(v_a)|$ *and* $y = |l(v_b)|$

Proof. Theorem 4.2.4 is effectively just a rewording of Theorem 4.2.3 because the operations they deal with are addition and deletion of an edge respectively which are reversal of each other. The first inequality of the Theorem 4.2.3 is the same as the second inequality of the Theorem 4.2.4. Similarly, the second inequality of the Theorem 4.2.3 is the same as the first inequality of the Theorem 4.2.4. Hence we provide a combined prove of both theorems. It is easier to prove the first inequality of Theorem 4.2.3 in the framework of edge addition while the second

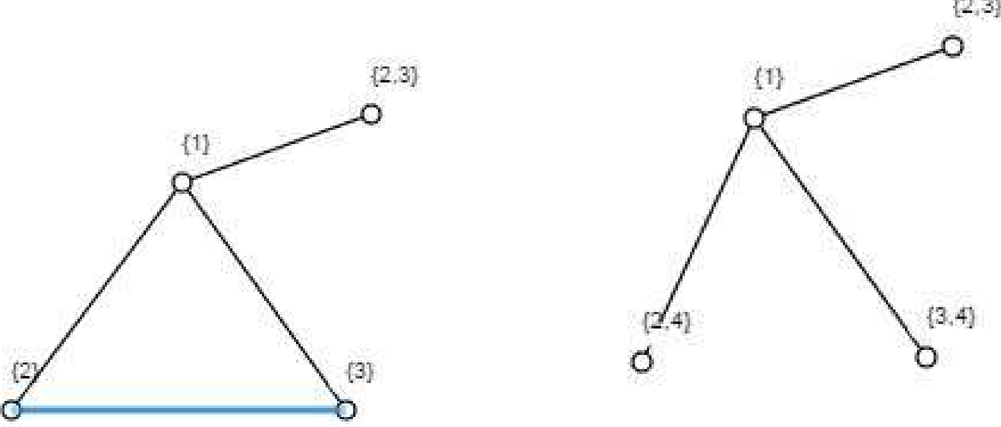


Figure 4.2: Lower bound on increase (decrease) in USN after removing (adding) an edge

inequality of Theorem 4.2.3 in the framework of edge deletion. Thus we follow this scheme.

Assume k elements are common between labels of v_a and v_b where $1 \leq k \leq min(|l(v_a)|, |l(v_b|)$

Procedure 1: Obtaining a valid labelling using k additional labels:

1. Compare the cardinalities of the labels of the two endpoints of the newly added edge. The label with higher cardinality is retained as it is.

2. Replace C=$\{1, 2, \ldots, k\}$, the common elements with C'=$\{1', 2', \ldots, k'\}$ in the label of the vertex which has lower cardinality.

3. For all other labels, if they contain any non-empty subset of C then add the corresponding non-empty subset from C' to their labels in order to preserve non-adjacency with the lower cardinality edge-endpoint.

After removal of an edge, USN may increase by at most one. This is because in the worst case, only one new element need be added in the labels of the endpoints in order to respect non-adjacency. □

It is possible to construct any given graph by considering the underlying complete graph on the same vertex set and removing the extra edges. Using this idea

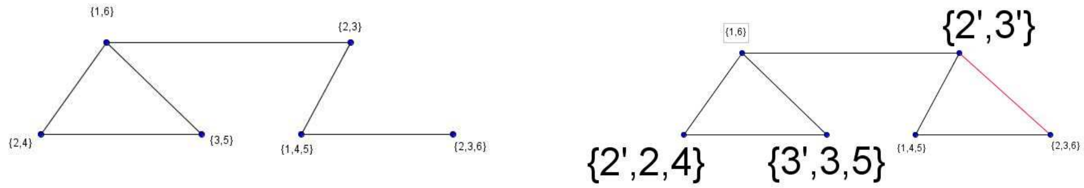


Figure 4.3: Upper bound on increase (decrease) in USN after adding (removing) an edge

in conjunction with Theorem 4.2.3, we derive universal upper bound on USN. This proof readily translates to an algorithm leading to generate a valid labelling respecting this upper bound.

Theorem 4.2.5. USN$(G) \leq (n-1)+\binom{n}{2}$.

Proof. We give an algorithmic proof for the claim.

Algorithm 1**: To find a valid labelling for any given graph**

Step 1: For any graph on n vertices start with the optimal valid labelling of the corresponding complete graph K_n with USN $= (n-1)$.

Total number of distinct elements used after this step is exactly $n-1$.

Step 2: Delete the necessary set of edges one by one from this K_n to transform complete graph into the given graph. After deleting each edge, add a new element to the labels of both endpoints of that particular edge.

Total number of additional elements used after step 2 is at most $\binom{n}{2}$ because the complete graph has exactly $\binom{n}{2}$ edges and in the worst case, all the edges are required to be deleted in order to construct the given graph.

Therefore, this algorithm gives a valid labelling of any graph with at most $\sum n-1$ elements. □

Theorem 4.2.6. USN $(K_n \cup K_n)$$= n^2$

Proof. Here, $K_n \cup K_n$ represent two disjoint copies of cliques. Each vertex of the first clique is non-adjacent to all the vertices of the second clique. In order to respect this non-adjacency, each vertex-label of the first clique must have at least one element common from each of the vertex-labels of the second clique. It is not possible to reuse elements in the second clique since all the vertices are adjacent

to one another. Hence, for each vertex present in the first clique, at least n distinct elements (one element from each vertex-label) are required from the second clique in order to respect adjacency in the first clique. Total number of vertices in the first clique are n. It is not possible to reuse elements in the first clique either. Hence, for a valid labelling at least n^2 elements are required (USN is at least n^2).

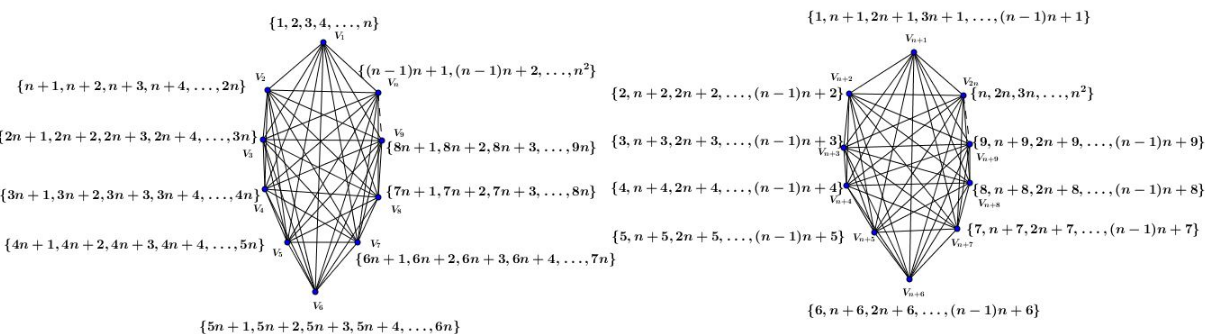


Figure 4.4: A valid and optimal labelling of two disjoint copies of K_n using n^2 elements

It is possible to assign valid labels using exactly n^2 elements by using following label assignment scheme.

For $1 \leq i \leq n$,

$L(V_i) = \{(i-1)n+1, (i-1)n+2, (i-1)n+3, \dots, in\}$

For $n+1 \leq i \leq 2n$,

$L(V_i) = \{(i-n), n+(i-n), 2n+(i-n), \dots, (n-1)n+(i-n)\}$

We have shown a valid labelling for two disjoint copies of clique using exactly n^2 elements and lower bound is also n^2. Hence USN is n^2. □

The final labelling is shown in Figure 4.4 which is valid and optimal.

As a corollary we have the following theorem.

Theorem 4.2.7. USN $((K_n \cup K_n) + e) = \dfrac{N^2}{4} - 1$ *where* $N = 2n$ *and* e *is the newly added edge.*

A tight upper bound on the intersection number (and hence on USN) is $\dfrac{N^2}{4}$ [14]. From Theorem 4.2.7, we can infer that limiting to connected graph does not give an asymptotic improvement on USN in general in the worst case.

4.3 Our proposed methods for graph visualization

We present a core ideas of two graph based methods to represent social networks. Both our proposed methods for graph visualization use the valid labelling obtained by algorithm described in the previous section as a starting point. Their actual application in social network is described in detail in the next section. In both these methods the vertices of the graph arising from the social application are placed on distinct grid points of the respective grids.

Method-1:

This method requires an n by n grid to represent the input graph on n vertices in which all labels are partitioned based on their cardinalities. (see Figure 4.9)

- Vertices of the graph are placed at distinct grid points of the aboved described grid.
- All nodes with label size i are placed in any order along distinct grid points of the line $x = i$ beginning from $y = 1$ and with no gaps.
- The largest y value used to represent a vertex along the line $x = i$ is the number of vertices whose labels have cardinality exactly i.
- Largest possible cardinality for any individual label is n (see Algorithm 1) . Therefore, the upper limit on the value of x-axis is n.
- At most n elements can be present in one group. Therefore, the upper limit on the value of y-axis is also n.

Method-2:

This method requires an n by $O(n^2)$ grid in which all labels are partitioned based on their individual label structures. (see Figure 4.12)

- Vertex-labels which are present in the xth column will have x^{th} element common in them. All the elements which are less than x are missing in these labels. i.e. $\{1, 2, \ldots, x-1\}$.
- All vertex labels whose least index element is i are placed in any order along distinct grid points of the line $x = i$ beginning from $y = 1$ and with no gaps.

- Value of y-axis represents the number of elements which are present in that group.
- Largest label generated by our method is USN is $O(n^2)$ (see Algorithm 1) . Therefore, the upper limit on the value of x-axis is $O(n^2)$.
- At most n elements can be present in one group. Therefore, the upper limit on the value of y-axis is n.

4.4 Application: Social network analysis

In this section, we describe how our proposed technique can be used for the analysis of a social network.

One sample graph which represents the friendship relationship between 6 people is shown in Figure 4.5 i.e. if edge (u, v) is present then person u and v are friends on social network sites like Facebook/Twitter. Corresponding complete graph is shown in Figure 4.6.

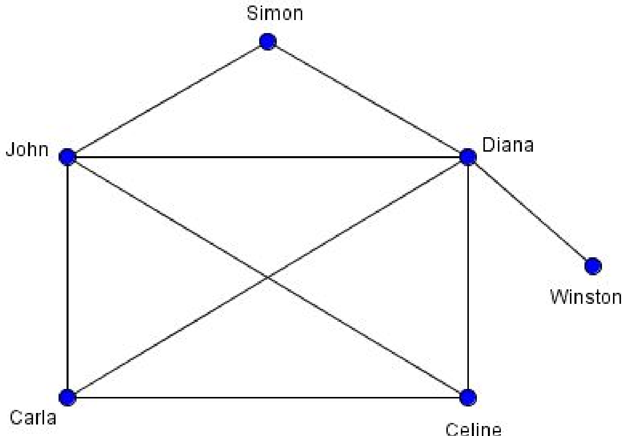


Figure 4.5: A) Social Network Graph

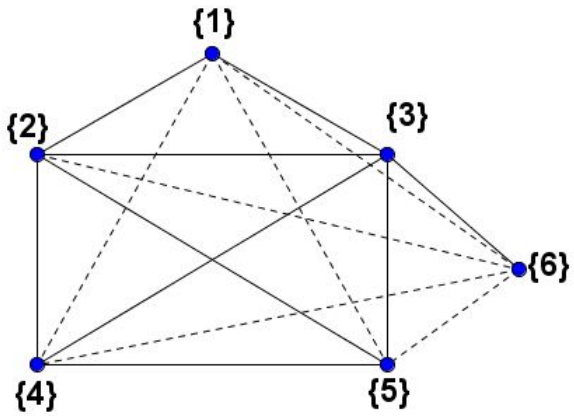


Figure 4.6: B) Corresponding complete graph

A few iterations of the Algorithm 1, for the construction of the valid labelling of the given graph are shown in the Figures 4.7 and 4.8.

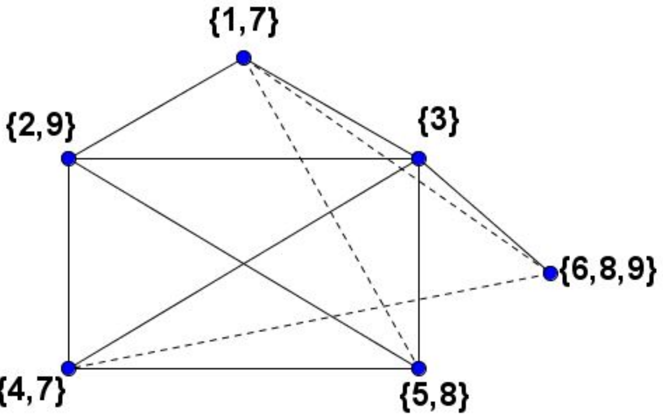


Figure 4.7: Steps for obtaining valid labelling of the Social Network Graph

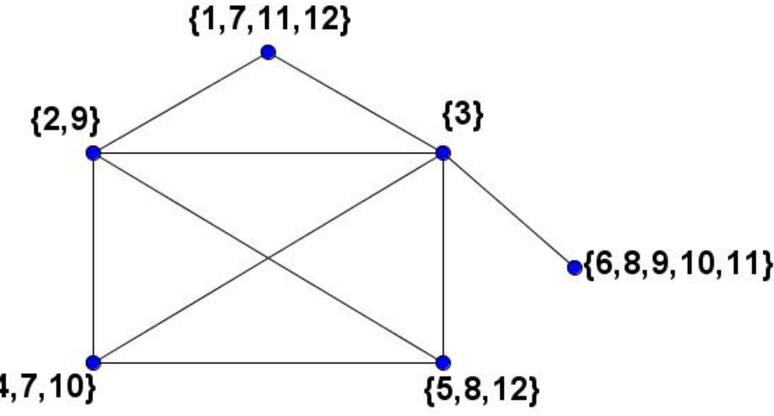


Figure 4.8: Steps for obtaining valid labelling of the Social Network Graph

Our proposed visualization method is shown in figure 4.9.

Key characteristics of our proposed method are listed below:

1. Space complexity: n by n grid is required for the visualization technique. This is because each vertex can be non adjacent to at most remaining $n-1$ vertices and therefore size of individual label can not exceed n (see Algorithm 1). Total number of vertices and hence total number of distinct labels are exaclty n. Therefore, total number of labels with the same cardinality can be at most n.

2. x-axis represents the cardinality of the individual label. i.e. classification is done based upon the size of the label. y-axis represents the number count corresponding to the each group.

3. It is easy to identify whether two nodes are related or not by comparing their corresponding labels. Nodes are related if and only if their corresponding labels are disjoint.

4. In the visualization, number of friends of a node tends to be inversely proportional to its individual label size. Here, label 3 has the minimum cardinality and therefore it is very likely that the corresponding person (Diana) has the most number of friends in the network which is actually true (see fig 4.5). Diana is a friend of all others.

5. Here, value of $|l|-1$ for any node denotes minimum number of non-neighbors for that particular node. For example, for $l = \{2,9\}$ value of $|l|-1$ is $2-1=1$ from which we can infer that John has at least one non-neighbour in the graph which is true. See Figure 4.5, John and Winson are not related.

6. Vertices which are present in the same vertical line have at least k non neighbors where $k = x-1$ where x denotes the value of x coordinate.

7. People who are present in first column tend to be the most popular whereas those who are present in the last column tend to be the least popular.

8. Identification of the clusters with same popularity in the underlying social network graph is possible using this visualization.

9. Further details can be obtained by assinging colors to the node. Here in our example, for female gender nodes, pink color is assigned whereas for males blue color is assigned. This will help to analyze gender specific patterns in the underlying social network graph.

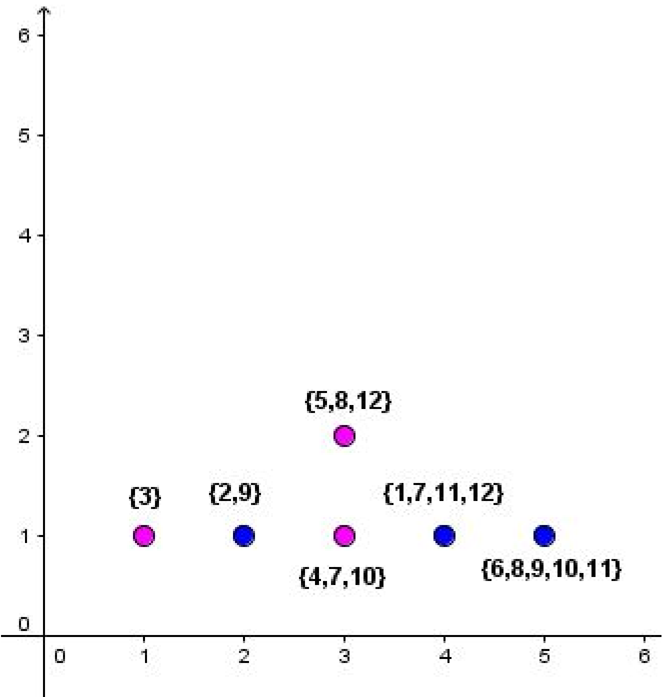


Figure 4.9: Edgeless Graph: Our proposed method

Using our proposed visualization method (edgeless graph), we will be able to analyse following crucial questions related to the given social network.

1. Identification of users who are most active as well as those who are most inactive. This is important for promotional activities.

2. It can help the social networking site to improve their "People you may know" feature.

3. With the use of labels it is easier to verify whether any given two persons are connected or not.

4. It is easy to identify all people with a specific number of friends.

4.4.1 Study of dynamic changes

Social networks are highly dynamic in the nature. Applying Theorems mentioned in Section 4.2, we can efficiently and effectively add (remove) new vertices and/or edges in our existing visualization.

In the Figure 4.10, modified graph is shown after adding 2 new vertices and 4 edges. The modified visualization is shown in Figure 4.11 which has following

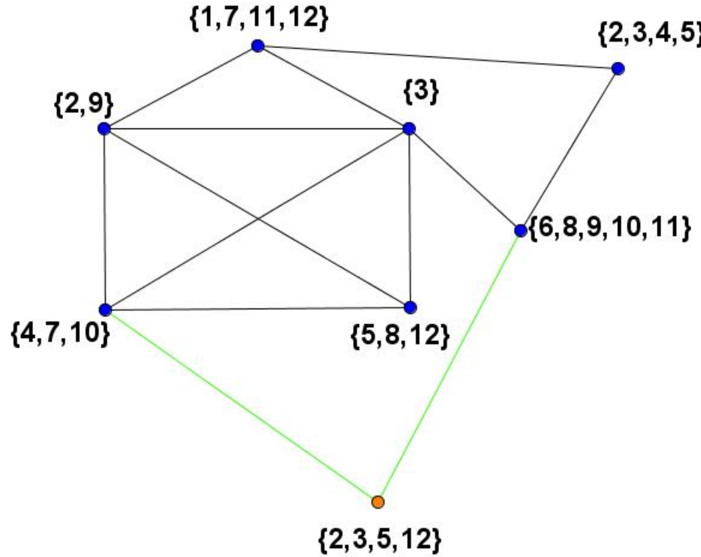


Figure 4.10: Valid labelling of the graph after addition of 2 new vertices and 4 edges

interesting features:

1. In this particular case, no additional space is required. Number of nodes is $n + 2$ but still the n by n grid is sufficient because no new elements were added during the procedure.

2. It is easy to identify the correct location for the newly added vertex based on its label size. Here both the newly added vertices have individual label size 4 and therefore they are placed in the 4th column.

3. One can also highlight the newly added vertices with some different color in order to compare their initial popularity in the underlying social network.

The visualization of the same graph (shown in Figure 4.10) using our proposed method-2 is shown in Figure 4.12.

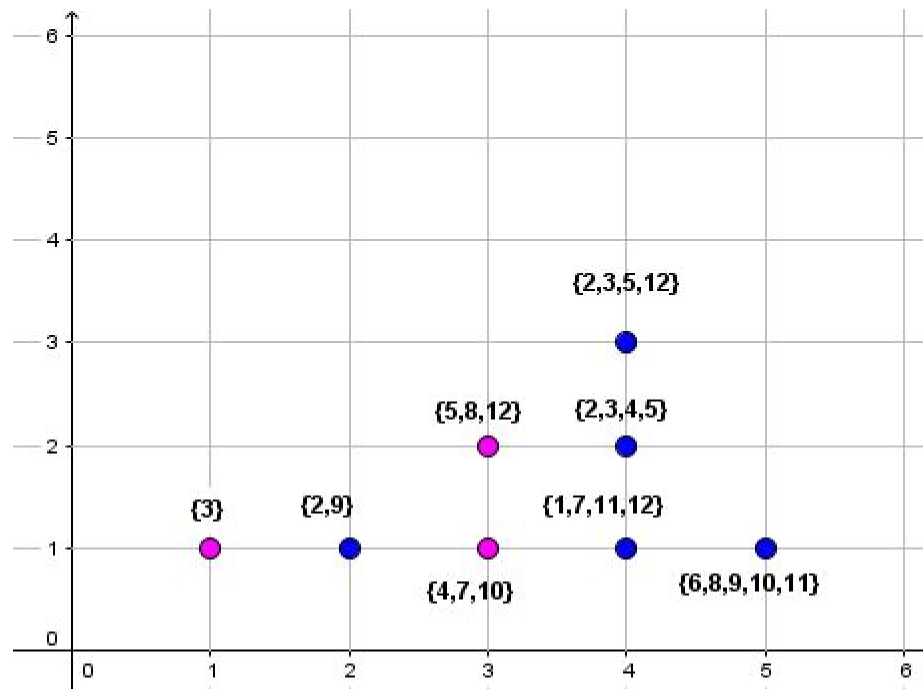


Figure 4.11: Representation of the graph after addition of 2 new vertices and 4 edges

The properties of this method are explained below:

- Elements which are vertically aligned have at least one element in common and therefore they are mutually non-adjacent.
- In order to identify, collection of mutually adajcent vertices (cliques), one can plot the valid labelling of the complement graph using this method.
- Space complexity: An n by $O(n^2)$ grid is required because size of individual label is at most n and the size of underlying labelling set is $O(n^2)$ (see Algorithm 1)
- Here, newly added vertices are represented using orange color and they are vertically aligned which says that the newly added vertices are non-adjacent to each other.

4.5 Conclusions and future Work

Our proposed data visualization techniques have almost all the necessary characteristics. It shows all the data without any information loss. Using our technique,

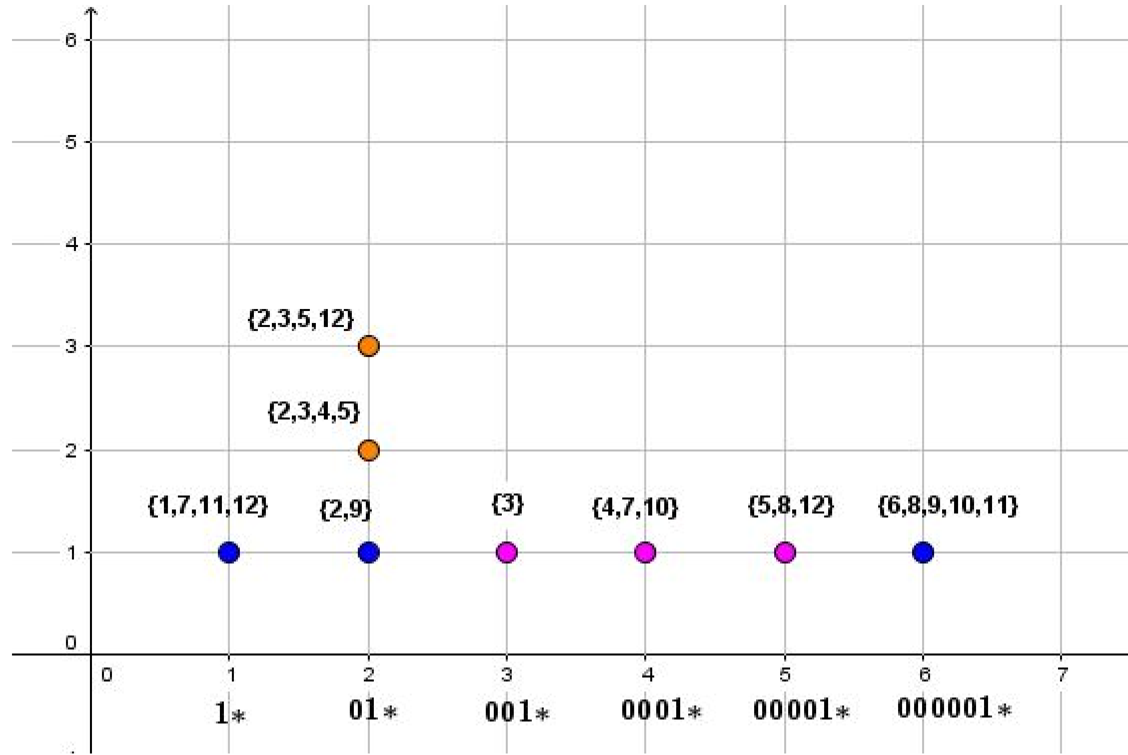


Figure 4.12: Proposed method-2 for identify collection of nodes who are mutually non-adjacent

it is possible to display large data sets coherently. The method encourages the human eye to compare different objects because each object is represented by a unique label.

In future, we plan to design a more efficient interactive version of the proposed techniques in which we would incorporate the following features: In the interactive version, we want to highlight edges which are present within certain vertical regions of the representation whenever the user clicks on it. Using this feature, users can understand whether those nodes which belong to the same cardinality group are related to each other or not. Within one vertical region of the representation, we will put disjoint labels i.e. adjacent neighbors within the alpha-neighborhood of them. This will help users to identify connections rapidly. Whenever the user clicks on a particular node, we will highlight all the neighbors of the node by doing real time computation on labels and we will also highlight the corresponding edges so that users can find out the neighbors.

CHAPTER 5

Results on UUSN, ILN and UILN

In Section 5.1 the problem statement is discussed along with some basic results. Results related to UUSN and ILN are discussed in detail for some specific classes of graphs (including complement of complete graph, matching, paths, cycles, complete bipartite graph, complete binary trees) in Section 5.2. Cartesian product based method is discussed in the same section. The final section summarises the results obtained in this chapter.

5.1 Introduction

A set labelling of a graph $G(V, E)$ is a function $f : V \rightarrow \mathcal{P}(\{1, 2, \ldots, k\})$ where $k \in Z^+$ such that

- f is one one.
- $\forall u, v \in V, (u, v) \in E \Leftrightarrow f(u) \cap f(v) = \phi$.

For a uniform set labelling following additional condition is required.

- $\forall u \in V, |f(u)| = c$ where $c \in Z^+$. c is the size of the sets in the optimal set labeling.

Uniform Universe Size Number (UUSN) of a graph is the least positive integer k such that a uniform set labelling of G exists (see Figure 5.1 for example) .

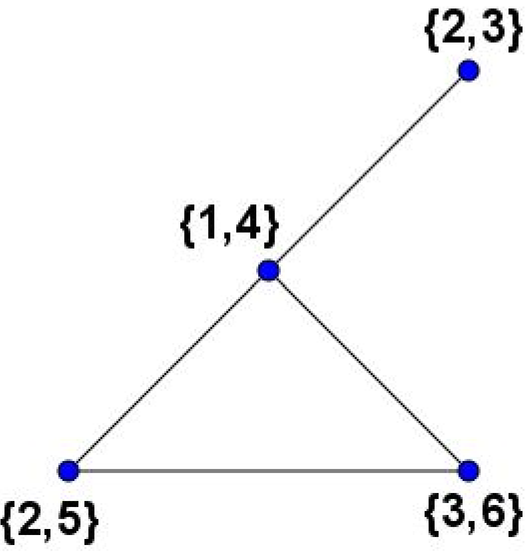


Figure 5.1: UUSN$(H) = 6$

Individual label number(ILN) of a graph is the smallest size of the largest label over all set labellings (not necessarily uniform) of the vertices with uniques sets such that adjacency coincides with disjointness (see Figure 5.2 for example).

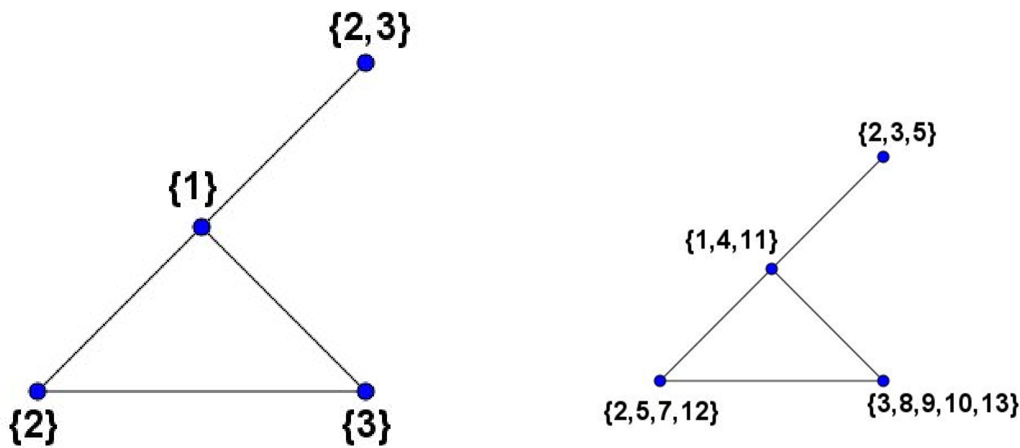


Figure 5.2: ILN$(H) = \min\{2, 5, \cdots\} = 2$

5.1.1 General results on UUSN and ILN

Here, we present general bounds on UUSN and ILN .

Theorem 5.1.1. ILN(G) <UUSN(G), *where G has at least* 2 *vertices.*

Proof. Consider a valid and uniform set labelling of any graph G, optimal in terms of the universe size. Clearly, the number of elements used in total is $\text{UUSN}(G)$. If the underlying set is used as label then exactly 1 label can be generated since it is impossible to use any of the subsets of the underlying set as a label of any other vertex (due to uniform cardinality constraint in this particular case). In this particular labelling, no vertex has more than $\text{UUSN}(G) - 1$ elements in its label. This proves the result. □

Theorem 5.1.2. $\text{UUSN}(G) \geq \lfloor log_2 n \rfloor + 1$ *and* $\text{ILN}(G) \geq 1$*, where G has n vertices.*

Proof. With the use of $\lfloor \log_2 n \rfloor$ elements, it is possible to generate at most $n - 1$ non-empty subsets. Using these subsets, at most $n - 1$ vertices can be labeled since repetitions of labels is not allowed. Here, $|V(G)| = n$ and therefore at least 1 additional element is required in order to assign non-empty as well as unique label to the nth vertex. Therefore value of UUSN is at least $\lfloor \log_2 n \rfloor + 1$ for any given graph.
Note: ILN $(K_1) = 0$.
The largest individual label size will be at least 1 in all possible set labellings of the given graph G where n is at least 2. □

Theorem 5.1.3. $\text{UUSN}(G + v) = \text{UUSN}(G) + c$*, if* $d(v) = n(G)$*. Here,* $\forall u \in V(G)$*,* $|f(u)| = c$ *and* $c \in Z^+$ *Here, c is the size of the sets in the optimal set labeling.*

Proof. Here the vertex v is adjacent to all other vertices and hence it is not possible to reuse any of the $\text{UUSN}(G)$ elements for the labelling of vertex v. In order to obtain uniform labelling for $G + v$, $|f(v)|$ must be c. Therefore, exactly c additional elements are required for obtaining uniform labelling of $G + v$ (see Figure 5.3). □

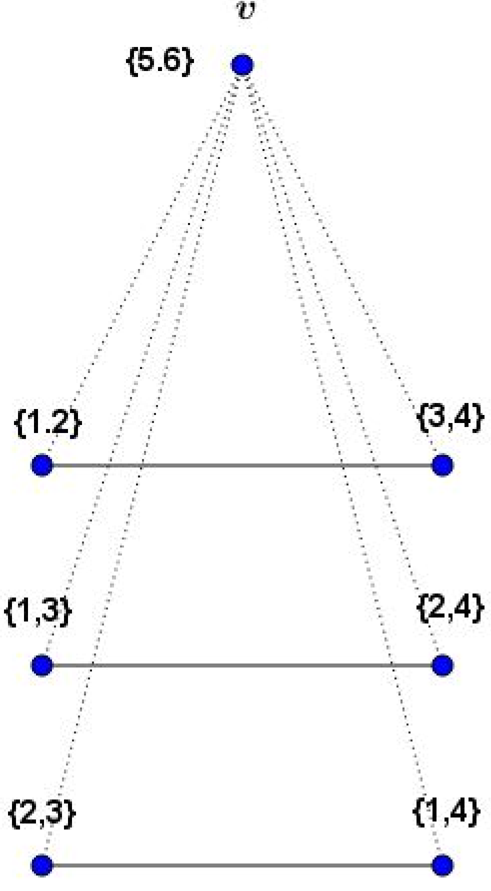


Figure 5.3: UUSN$(M_6) = 4$ and UUSN$(M_6 + v) = 6$

Theorem 5.1.4. UUSN$(K_n) = n$

Proof. UUSN$(K_1) = 1$ and from the application of Theorem 5.1.3 exactly $(n-1)$ times iteratively starting with K_1. □

Theorem 5.1.5. ILN$(G + v)$ =ILN(G)*, if* $d(v) = n(G)$.

Proof. Here the vertex v is adjacent to all other vertices and hence it is not possible to reuse any of the k elements used for valid set labelling for the labelling of vertex v. In order to obtain a valid set labelling for $G + v$, it is sufficient to assign a singleton set {k+1} as a label for vertex v. ILN(G) is at least 1 (from Theorem 5.1.2) and hence value of ILN won't change for $G + v$. □

Theorem 5.1.6. ILN$(K_n) = 1$

Proof. $\text{ILN}(K_2) = 1$ and from the application of Theorem 5.1.5 exactly $(n-2)$ times iteratively starting with K_2. □

Theorem 5.1.7. UILN*(G)*=ILN*(G)*

Proof. Let $\text{ILN}(G) = k$. i.e. there exists at least one labelling of G where the cardinality of the highest individual label is k and cardinalities of all other labels are less than or equal to k. Let $l(v_i)$ denote the label of vertex V_i. By adding $k - |l(v_i|$ additional elements in each individual vertex label, it is possible to obtain a uniform labelling where cardinality of each individual label is still k which proves the result. □

The following Theorem gives the universal upper bounds on UUSN and ILN.

Theorem 5.1.8. $\text{UUSN}(G) \leq n+ \binom{n}{2}\ (n-1)$ *and* $\text{ILN}(G) \leq n$.

Proof. We give an algorithmic proof for the claim.

Algorithm 1: To find a valid uniform labelling for any given graph

Step 1: For any graph on n vertices start with the optimal valid labelling of the corresponding complete graph K_n with $\text{UUSN} = n$.

Step 2: Delete the necessary set of edges one by one from this K_n to transform complete graph into the given graph.

After deleting each edge, the following operations are performed:

Step 2(a): Add an extra element to labels of both endpoints of that particular edge (to establish non-adjacency between endpoints).

Step 2(b): In order to obtain uniform labelling, add exactly 1 extra element in the labels of all $(n-2)$ vertices excluding the endpoint vertices considered in Step $2(a)$. So during the deletion of each edge a total of $(n-1)$ new elements are required including the element which is added in the labels of the two endpoints of the deleted edge.

Total number of distinct elements used after step 1 is exactly n.

Total number of additional elements used after steps $2(a)$ and $2(b)$ is at most $\binom{n}{2}(n-1)$ because complete graph has exactly $\binom{n}{2}$ edges and in the worst case, all the edges are required to be deleted in order to construct the given graph. Therefore, $\text{UUSN}(G) \leq n+ \binom{n}{2}\ (n-1)$.

For the upper bound calculation of ILN, consider the step 1 and $2(a)$ only of Algorithm 1. Here, after step 1 $\text{ILN}(G)$ is 1 (Theorem 5.1.6). Step $2(a)$ can be applied at most $n-1$ times for each vertex and each iteration of step 2a) will increase the $\text{ILN}(G)$ by at most 1. Therefore, $\text{ILN}(G) \leq n$. □

Theorem 5.1.9. $\text{UUSN}(G) \leq T + n{*}\text{ILN}(G) - \text{USN}(G)$

Proof. Consider a valid set labelling of G. Let the cardinality of the underlying labelling set be T and $\text{ILN}(G)$ be the cardinality of the largest label size in the labelling. Note that T is not necessarily equal to $\text{USN}(G)$.

In order to form a uniform as well as valid labelling of G, cardinalities of each vertex label must be same. Total number of elements required for obtaining a valid and uniform labelling can never be more than $n{*}\text{ILN}(G)$.

Let us say, cardinality of vertex label v_i is c_i. So, in addition to T, we need at most $n{*}\text{ILN}(G) - \sum_{j=1}^{n} c_j$ elements for obtaining a valid and uniform labelling. One can observe that $\sum_{j=1}^{n} c_j \geq \text{USN}(G)$ since each element of the underlying uni-

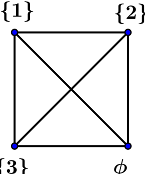


Figure 5.4: $\text{ILN}(G) = 1$, $\text{USN}(G) = 3$, and $\text{UUSN}(G) = 3 + 4 * 1 - 3 = 4$

verse set is used at least one time in vertex-labels.

So we need in total $T + n{*}\text{ILN}(G) - \text{USN}(G)$ elements for obtaining a valid and uniform labelling in the worst case. □

5.2 Results on UUSN and ILN for some special families of graphs

In this section, we derive results (either exact or asymptotic) on UUSN and ILN on the following classes of graphs: Paths, Cycles, Wheel Graph, Complement of Complete Graph $(\overline{K_n})$, Matching (M_{2n}), Complete binary trees, Complete bipartite graphs.

Theorem 5.2.1. *For matching,* UUSN $= O(\log_2 2n)$ *and* ILN $= O(\log_2 2n)$

Proof. These results are direct consequences of Theorem 2.2.1. □

Theorem 5.2.2. UUSN$(\overline{K_n}) \leq 1.5(1 + \lceil \log_2 n \rceil)$ *and* ILN$(\overline{K_n}) = 2$

Proof. Consider A as the underlying universal set with cardinality k. Here disjoint subsets are not allowed for labelling of independent set. For all S (where $S \subset A$), at most 1 subset can be used for labelling from each pair $(S, A - S)$. So the total available sets for labelling are reduced by factor of half. Select the subset having higher cardinality in each pair and if cardinalities are same then make an arbitary selection. By doing this, each subset will have cardinality at least $\frac{k}{2}$. All these selected subsets will have non-empty intersection (pigenhole principle) and hence they can be used to assign labels to vertices.
Note: Empty set won't be used in labelling since in the pair (ϕ, A), A has higher cardinality.
So, in summary total at most 2^{k-1} vertices of the graph can be labelled using A. In order to obtain uniform labelling, at most $\frac{k}{2}$ additional elements are required since the least possible cardinality is $\frac{k}{2}$ and greatest cardinality is k (since A is used as label and $|A| = k$). So after adding at most $\frac{k}{2}$ elements in each label, all the labels will have uniform cardinality of k. Therefore, the total number of elements required is $1.5k$ in order to obtain a uniform labelling of 2^{k-1} vertices.

So using $1.5k$ elements, it is possible to do obtain a valid and uniform labelling of $(\overline{K_n})$ where $n \in \{2^{k-2}+1, 2^{k-2}+2, \cdots, 2^{k-1}\}$ which proves the result for UUSN.

ILN is at least 2 for $\overline{K_n}$. This is because at least 1 element must be common in the labels of every vertex pair and in order to avoid repetition of labels, at least one more element is required to be added in each of the individual labels.

Valid-ILN-labelling-$\overline{K_n}$

1. For each $v_i \in \overline{K_n}$ assign $\{i\}$ as its label.

2. Add $\{i+1\}$ in the labels of all $v_i \in \overline{K_n}$.

After the 2nd step, the cardinality of each individual label is 2 which proves the result for ILN. □

Theorem 5.2.3. UUSN $(K_{s,t}) \leq 1.5(2 + \lceil \log_2 s \rceil + \lceil \log_2 t \rceil) + |\lceil \log_2 s \rceil - \lceil \log_2 t \rceil|$
ILN $(K_{s,t}) = 2$

Proof. Complete bipartite graph consists of two independent sets. From the valid and uniform labelling of one independent set nothing can be used in the second independent set. So labelling of these two independent sets must be entirely disjoint. Now cardinality of each vertex label in first partite set is $1 + \lceil \log_2 s \rceil$ whereas cardinality of each vertex label in second partite set is $1 + \lceil \log_2 t \rceil$. In order to make cardinality of each label same, $|\lceil \log_2 s \rceil - \lceil \log_2 t \rceil|$ must be added to all the vertex labels to one of the smaller partite set (the partite set which has smaller individual label size for all vertices). This proves the result for UUSN. From Theorem 5.2.2, it is possible to label both partite sets with ILN $= 2$ and ILN will remain 2 for the whole graph if disjoint sets are used for the labelling of both the partite sets. This proves the result for ILN. □

Theorem 5.2.4. UUSN$(P_n) = O(logn)$.

Add-edge Procedure:
Input: A valid labelling of any given path (P_n) using exactly k labels.
Output: A valid labelling of P_{n+2} using exactly $k+3$ labels.

Step 1: Identify a P_4 in the given P_n. Let the vertices of P_4 be v_r, v_{r+1}, v_{r+2} and v_{r+3} and corresponding labels be l_r, l_{r+1}, l_{r+2} and l_{r+3} respectively.

Step 2: Add 2 new vertices v_{n+1} and v_{n+2}. In order to construct P_{n+2}, add 3 new

edges (v_{r+2}, v_{n+1}), (v_{n+1}, v_{n+2}), (v_{n+2}, v_{r+3}).

Step 3: $l(v_{n+1}) = l_{r+1}$

$l(v_{n+2}) = l_{r+2}$

Step 4: Three pairs (v_r, v_{n+1}), (v_{r+1}, v_{n+2}) and (v_{r+2}, v_{r+3}) are non-adjacent. In order to reflect non-adjacency in the labelling, add 3 distinct new elements say a, b and c to the labels of these 3 pairs respectively. The final labelling is valid. It respects adjacency as well as non-adjacency for all pairs of vertices.

Lemma 5.2.5. *For the given P_n, the Add-edge procedure can be applied for at most $\frac{n-1}{3}$.*

Proof. Add-edge procedure can only be applied to each edge-disjoint P_4-subgraph of the given P_n. The maximum number of edge-disjoint P_4 in a P_n is $\frac{n-1}{3}$. Thus the result follows. □

Proof of Theorem 5.2.4: We now give an algorithmic proof.

Algorithm-Valid-Uniform-Pathlabelling:

Input: A valid and uniform labelling of P_n using exactly k labels.

Output: A valid and uniform labelling of P_{n+2i} using exactly $k+3$ labels where $0 < i \leq \frac{n-1}{3}$ and $i \in N^+$

Step 1: Apply the Add-edge procedure on the given P_n for the first 4 vertices i.e. $r = 1$.

Step 2: For $j \geq 1$, (where $j \in N^+$)

If $3j+1 < n$ and $3j+4 \leq n$ then apply the Add-edge procedure for $r = 3j+1$ with the following modifications in step 4 of the procedure:

(Observation: v_{3j+1} participates in exactly two Add-edge procedures, in iteration $j-1$ as well as iteration j. Therefore, before starting of iteration j, the label of v_{3j+1} already has an additional element p due to iteration $j-1$ where $p \in \{a, b, c\}$. For $j = 1$, $p = \{c\}$ from step 1. The label of v_{3j+1} won't change during the jth iteration and $p \in l_{3j+1}$.)

1. Add p in the label of newly added vertex v which is a neighbour of v_{3j+3}.

2. Use remaining two elements $\{a, b, c\} - \{p\}$ in any order, for the remaining two pairs.

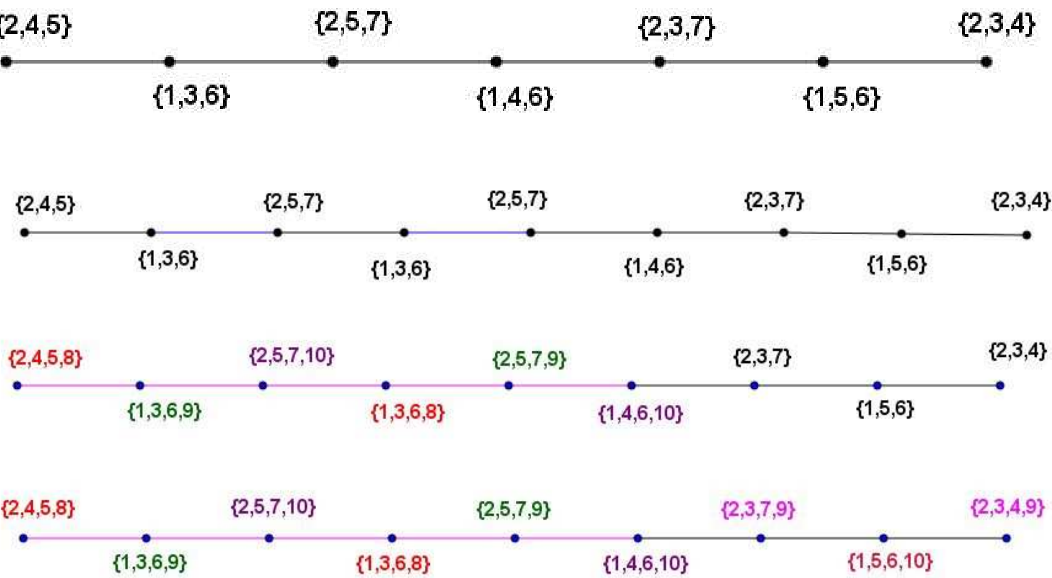


Figure 5.5: UUSN labelling- P_7 to P_9

Step 3: After j^{th} iteration of the Addedge procedure, the cardinality of the labels of the first $3j+1$ vertices will be increased by exactly 1. So the first $3j+1$ vertices of the path has a uniform labelling. The label of the $(3j+1)^{th}$ vertex will certainly contain exactly one of the 3 elements namely a, b, c. WLOG a is used in the label of the $(3j+1)^{th}$ vertex.

Step 4: Partition the remaining vertices $V \setminus \{V_1, V_2, \ldots, V_{3j+1}\}$ into two sets:
A: Even numbered vertices and
B: Odd numbered vertices.
Add b to all labels of A and add c to labels of B. After this step, the cardinality of the remaining vertices are also increased by 1. In general, the cardinality of each vertex is increased by exactly 1. So the final labelling is uniform and valid. No additional elements are required in this procedure.

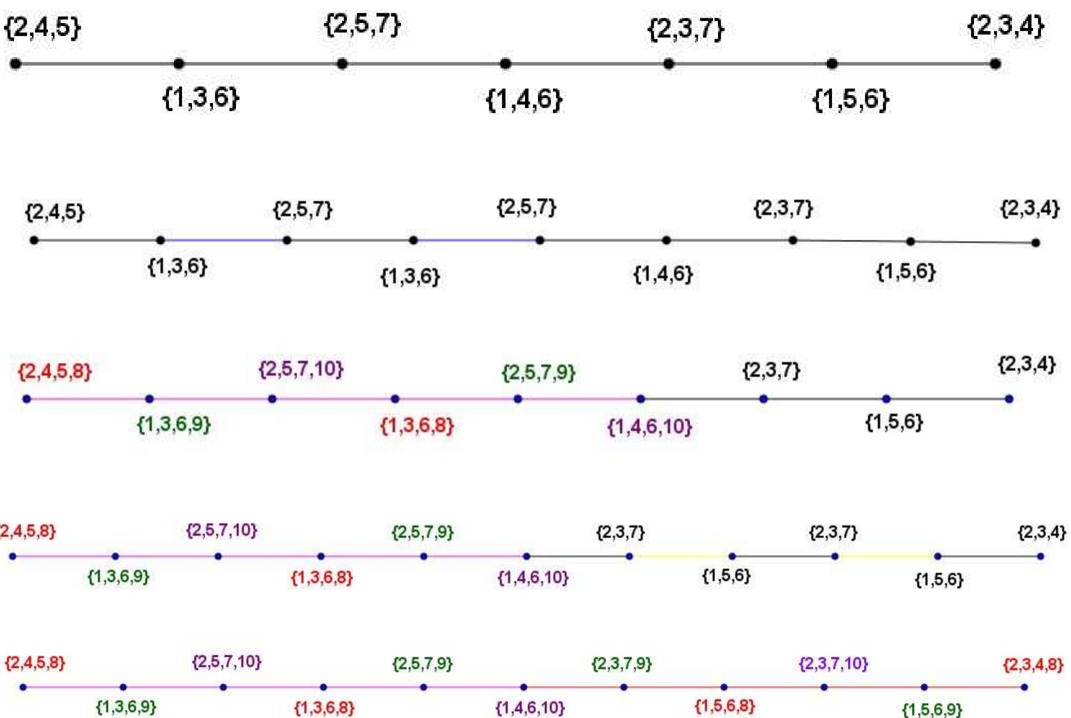


Figure 5.6: UUSN labelling- P_7 to P_{11}

Examples of constructions are shown in Figures 5.5 and 5.6.

Upper bound calculation:

So at most $\left\lfloor \frac{n-1}{3} \right\rfloor$ edges can be used and they will generate at most $\dfrac{2n-2}{3}$ new vertices.

Recurrence: $T(5n/3) = T(n) + 3$

UUSN : $\mathrm{O}(\log n)$ $\square$

Note: We can generate paths for all values of n by applying the Add-edge procedure repeatedly. One can generate all P_{2x+1} by repeatedly applying the Add-edge procedure on a valid and uniform labelling of P_7, similarly all P_{2x} can be generated by application of the Add-edge procedure on a valid and uniform labelling of P_6. Here $x > 3$ and $x \in \mathcal{N}$.

Theorem 5.2.6. *UUSN$(C_n) = O(\log n)$. and UUSN$(W_n) = O(\log n)$.*

Proof. With the use of the Valid-Uniform-Pathlabelling algorithm, it is possible to generate valid and uniform labelling of C_n using $O(\log n)$ labels. Notice that the edge (V_1, V_n) does not participate in any of the Add-edge procedure iterations.

Wheel graph consists of C_n and one additional vertex with degree n. Therefore, $\text{UUSN}(W_n) = \text{UUSN}(C_n) + c$. (from Theorem 5.1.3) □

Theorem 5.2.7. $\text{ILN}(P_n) = O(logn)$.

Proof. Old-ILN: Value of ILN before applying procedure Add-edge.
Procedure Add-edge may increase the cardinality of individual labels by at most one. Hence New-ILN= Old-ILN+ 1. If we obtain P_n using the Add-edge procedure, then $\text{ILN}(P_n) = \text{UUSN}(P_n)/3$. So ILN($P_n$)= $O(\log n)$. □

The same idea is also applicable on cycles as well as wheel graph.

Theorem 5.2.8. ILN(C_n) = $O(logn)$ *and* ILN(W_n) = $O(logn)$.

Theorem 5.2.9. $\text{UUSN}(BT_n) = O(logn)$ *and* $\text{ILN}(BT_n) = O(logn)$. *Where BT=Complete binary tree and n denotes the total number of vertices of the BT.*

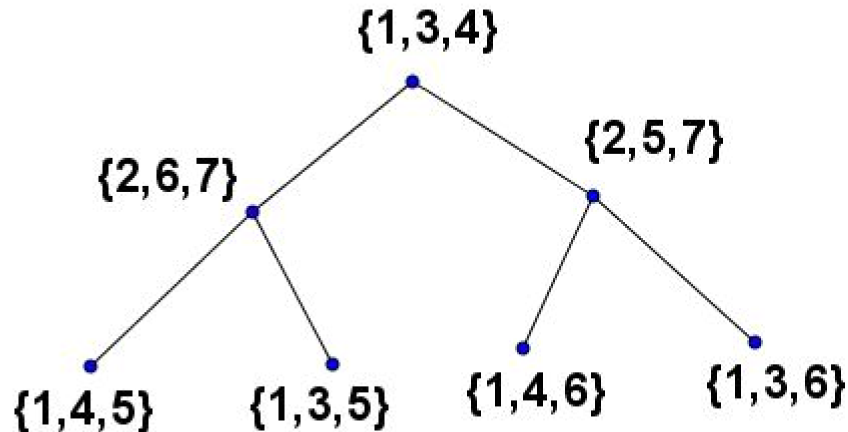


Figure 5.7: Input

Proof. We prove this result by an iterative algorithm: Valid-Uniform-TreeLabelling. The algorithm is explained below:
Input: A valid as well as uniform labelling of the complete binary tree of height h using exactly K labels.
Base case: $\text{UUSN}(BT_7) = 7$. Base case is shown in Fig. 5.7 with underlying labelling set: $\{1, 2, 3, 4, 5, 6, 7\}$.

Output: A valid and uniform labelling of complete binary tree of height $h+1$ using exactly $K+10$ labels. The ten new labels are a,b,c,d,e,f,g,h,i,j.

Step 1: For all newly added vertices in level $L+1$, find their corresponding ancestors in level $L-1$.
For all v (where v is a level $L+1$ vertex) ,
Label(v)= Label(ancestor(v) in level $L-1$).
Note: The levels $L+1$ and $L-1$ are highly similar with respect to adjacency with the remaining levels. Both of these levels are adjacent to level L and non-adjacent to $1, 2, 3, \ldots, L-3$.

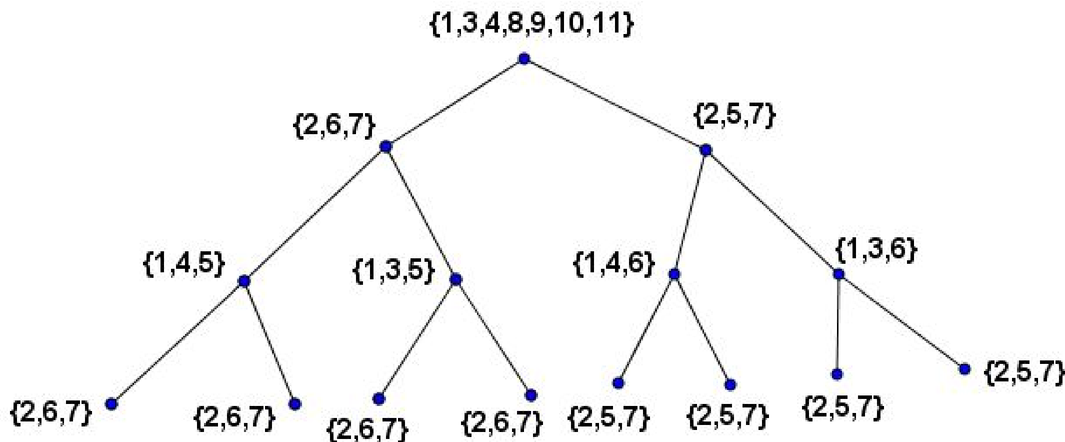


Figure 5.8: After Step 1 and 2

Observation after step 1:

- All the vertex-labels which are present in level $L+1$ will have non-empty intersection with each other because corresponding ancestors are either same or having non-empty intersection. All vertices present at level $L-2$ forms an independent set and since the underlying labelling is valid, all the vertex-labels pairs have non-empty intersection. So the vertex-labels which are present in level $L+1$ have non-empty intersection with each other because they are generated from the vertex labels present at $L-2$. This is desirable because all vertices of level $L+1$ are non adjacent.

- Layer $L+1$ and $L-1$ are non adjacent and they have non-empty intersection after this step.

Step 2: The level $L-2$ is adjacent to $L-1$ but not to $L+1$.
For all v', (where v' is a level $L-2$ vertex)
$New\text{-}Label(v') = Old\text{-}Label(v') \cup \{a, b, c, d\}$ (See Figure 5.8)

Step 3: Consider the sequential ordering (left to right) of vertices which are present in level $L+1$.
For each vertex v_i,

$$\begin{aligned} New\text{-}Label(v_i) &= Old\text{-}Label(v_i) \cup \{a\} \text{ if } i \mod 4 = 1 \\ &= Old\text{-}\ Label(v_i) \cup \{b\} \text{ if } i \mod 4 = 2 \\ &= Old\text{-}\ Label(v_i) \cup \{c\} \text{ if } i \mod 4 = 3 \\ &= Old\text{-}Label(v_i) \cup \{d\} \text{ if } i \mod 4 = 0 \end{aligned}$$

Vertices of level $L-2$ and $L+1$ will preserve non-adjacency because the corresponding labels have non-empty intersection after this step.

Step 4: Consider the sequential ordering (left to right) of vertices which are present in level $L : v_1, v_2, \ldots, v_q$
For each vertex v_i,

$$\begin{aligned} New\text{-}Label(v_i) &= Old\text{-}Label(v_i) \cup \{c, d\} \text{ if } i \mod 2 = 1 \\ &= Old\text{-}\ Label(v_i) \cup \{a, b\} \text{ if } i \mod 2 = 0 \end{aligned}$$

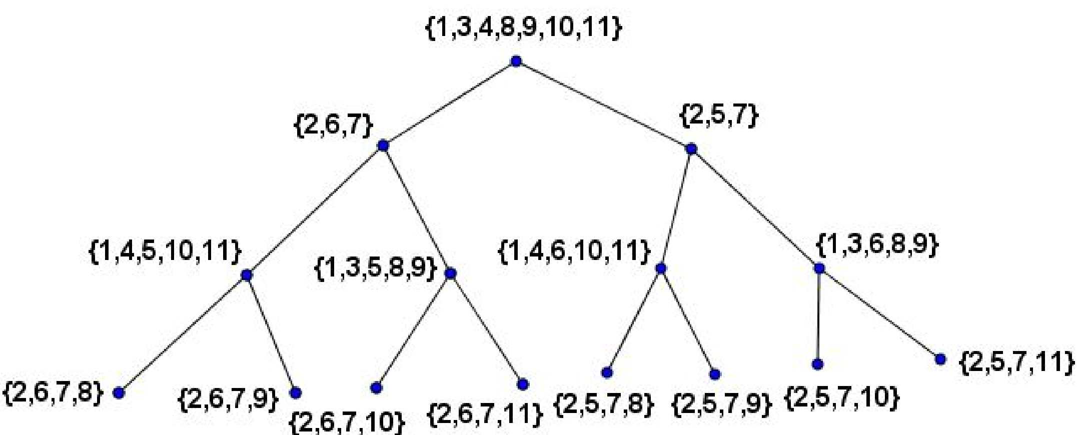


Figure 5.9: After Step 4

Observation after step 4:

Level $L+1$ and L preserves adjacency as well as non-adjacency.

The labelling after this step is valid and preserve adjacency as well as non-adjacency (see Figure 5.9)

Step 5: The following changes are required in order to obtain a uniform labelling:

1. Add $\{e, f, g, h\}$ to the labels of all the vertices of levels $(L-1)$, $(L-3)$, $(L-5)$, $\ldots$, 1. (if L is even).

2. Add $\{a, b, c, d\}$ to the labels of all the vertices of levels $(L-4)$, $(L-6)$, $(L-8)$, $\ldots$, 2. (if L is even).

3. Add $\{i, j\}$ to the labels of all the vertices of level L.

4. Add $\{e, f, g\}$ to the labels of all the vertices of level $L+1$.

Note: If L is odd then consider levels $(L-1)$, $(L-3)$, $(L-5)$, $\ldots$, 2 for step 5.1 and $(L-4)$, $(L-6)$, $(L-8)$, $\ldots$, 1 for step 5.2.

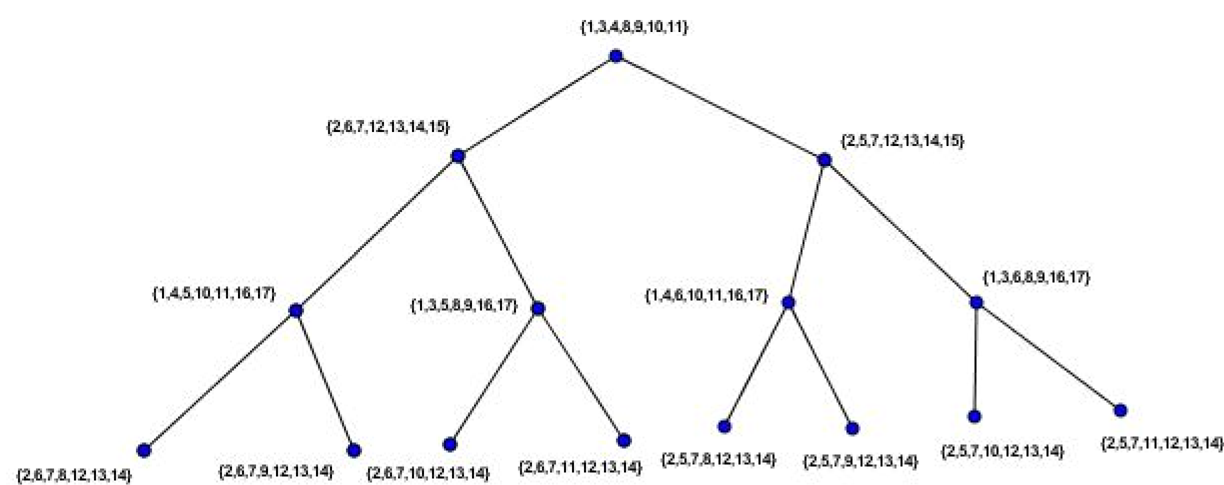


Figure 5.10: Output

The final output is shown in Figure 5.10.

For the complete binary tree, $n = 1 + 2 + 4 + 2^h = 2^{h+1} - 1$ i.e. $h = O(logn)$. For valid and uniform labelling of each layer exactly 10 additional elements are

required. Total number of layers are $O(logn)$. Therefore total number of elements required are $10 * O(logn)$ which is $O(logn)$.

Size of individual label is increased by exactly 4 after each iteration and total number of iterations are $O(logn)$. Therefore, ILN is $4 * O(logn)$ which is $O(logn)$.

□

Theorem 5.2.10. UUSN$(Q_n) < 3n + O(logn)$

Proof. The hypercube labelling algorithm, adds exactly n elements in each label (except for labels of the first two layers of the hypercube). If the underlying path on $(n+1)$ vertices has a valid uniform labelling with $O(\log n)$ elements then it is possible to obtain a final uniform labelling using $3n + O(\log n)$ labels using following the modified algorithm.

1. Apply hypercube algorithm (see Theorem 2.2.6) to get a valid non-uniform labelling (non uniform just because of the first two layers). Here use valid and uniform labelling of the underlying path graph to generate a valid labelling of the hypercube (see Figure 5.11).

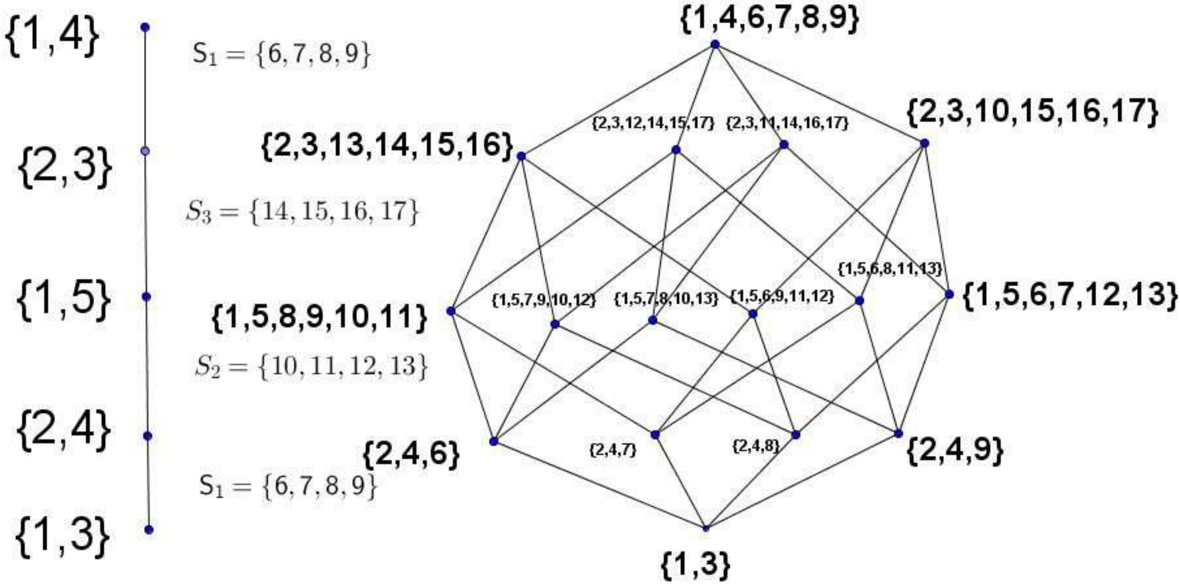


Figure 5.11: Labelling after step 1

2. The first layer contains only a single vertex say v. New label of v = old label of v $\cup$ S_k, where S_k is underlying set for labelling which is used in the 3rd layer of the hypercube.

3. Labels of the second layer contains only one additional element apart from the elements of the 2nd vertex of the corresponding path. Consider the $(n-1)^{th}$ layer of the hypercube which contains all $(n-1)$ sized subsets of the underlying labelling set. Layer 2 and $(n-1)$ both contains the same number of elements. Add all $(n-1)$ element subsets generated at layer $(n-1)$ into their corresponding copy in the 2nd layer.

The final labelling is valid and uniform (see Figure 5.12). $\square$

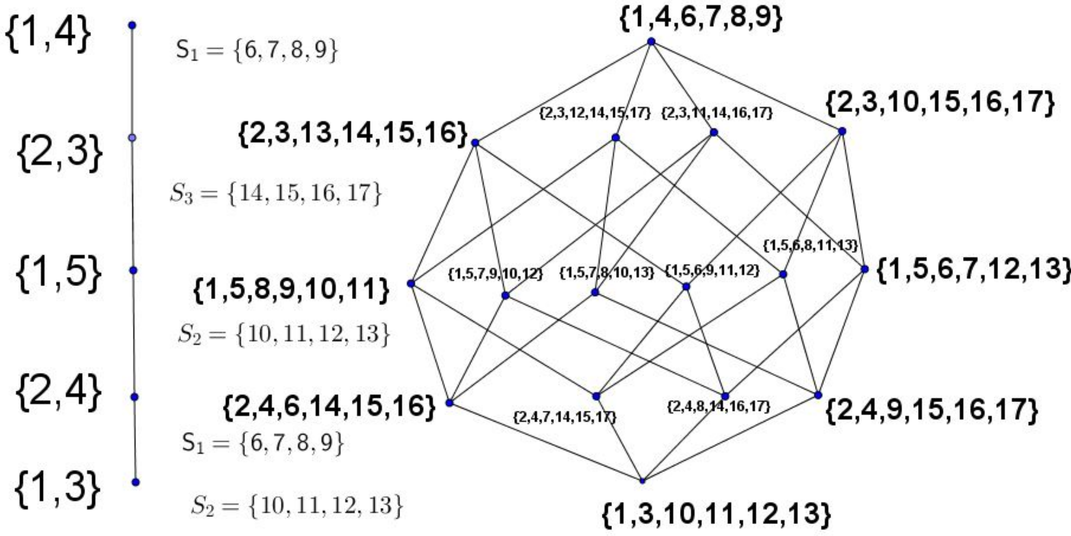


Figure 5.12: Uniform labelling of Q_4

Theorem 5.2.11. $\text{ILN}(Q_n) < n + O(logn)$

Proof. After applying hypercube labelling algorithm, cardinality of each label will be at most n+O(logn). Hence individual label number is at most $n + O(logn)$. $\square$

5.2.1 Cartesian product based method

Key Observation: Let A, B, C, D be sets. Then $(A \times B) \cap (C \times D) \neq \emptyset$ if and only if $A \cap C \neq \emptyset$ and $B \cap D \neq \emptyset$.

Theorem 5.2.12. *Let G and H be two graphs on the same vertex set V. Further, suppose $E(G) \cap E(H) = \emptyset$. Then* $\text{UUSN}(G + H) \leq \text{UUSN}(G) \times \text{UUSN}(H)$ *and* $\text{ILN}(G + H) \leq \text{ILN}(G) \times \text{ILN}(H)$.

Proof. Take optimal labellings of the vertex set using disjoint universes for the graphs G and H. Now consider the graph $G + H$. The vertex sets of G and H are identical. For each vertex v in $G + H$ give it the label $l_G(v) \times l_H(v)$. Clearly two vertices are nonadjacent only if they are nonadjacent in both G and H. In that case their labels under the two labellings will each be intersecting. From the key observation, it follows that their cartesian product new label will also intersect. Similarly for the case of non-intersection (adjacent vertices). □

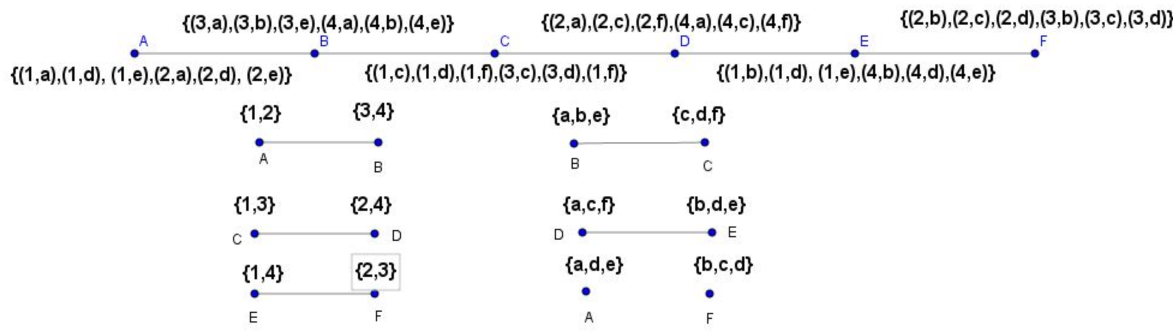


Figure 5.13: Cartesian Product Based Method

As a corollary we have the following theorem.

Theorem 5.2.13. $\text{UUSN}(P_n) \leq (1 + \log \frac{n}{2})^2$.

Proof. The path is the union of two disjoint matchings. Each matching has UUSN $O(\log n)$. From Theorem 5.2.12, we see that the graph has UUSN $O(\log n)^2$. We state this here just as an application since we have a better bound for paths (Theorem 5.2.4). □

Generation of valid and uniform labelling of P_6 using Theorem 5.2.13 is shown in Figure 5.13.

5.3 Conclusions and future Work

The work presented in this chapter has been published by us. The reference is [33].

We have obtained upper bound of UUSN and ILN for the complement of complete graphs, complete graphs, complete bipartite graphs, paths, cycles, matching, wheel graph, hypercube. In the future we plan to derive optimal and/or lower bound results for hypercube, harary graph etc.

CHAPTER 6

Total Graphs

The chapter is organised as follows. Background details related to total graphs are presented in Section 6.1. The detailed definitions of structures as well as notation is presented in Section 6.2. Basic properties and theorems on total graphs are presented in Section 6.3. Section 6.4 presents our results for complete graphs. In Section 6.5 we present our theorems characterising the two classes of vertices and develop our theorem into an algorithm for reconstructing the inverse total graph of a given total graph. We summarise our work and indicate possible future directions for research in Section 6.6.

6.1 Introduction

We obtain a new characterisation of total graphs based on the induced subgraphs on the neighbourhood of maximum degree vertices. These characterisations allow us to distinguish vertex-vertices (the vertices of the original graph) from edge-vertices (the vertices of the line graph) among the vertices of maximum degree. We also rely on the preponderance of maximal triangles (maximal cliques on exactly three vertices) between the vertex graph and the line graph consisting of two vertices from the vertex part and one vertex from the line graph part. Using this characterisation, we develop an efficient algorithm which iteratively creates the partition of the vertex set of the candidate total graph into its inverse total graph and line graph.

6.1.1 Related work

The notion of total coloring was introduced by Behzad [9] and Vizing [60] and those papers also conjectured that $\chi_T(G) \leq \Delta(G) + 2$. It is immediate that $\chi_T(G) \geq \Delta(G) + 1$, since a vertex of maximum degree and its incident edges must all get distinct colours. A lot of work has been done on total coloring [28] [8], based on frugal coloring [27], the list coloring conjecture [51] etc.

Examples of work on characterising graph classes include planar graphs [44], line graphs [39], interval graphs [21], bipartite graphs [3], graph factoring under the cartesian product operation [29].

6.1.2 Total graphs and set labelliing

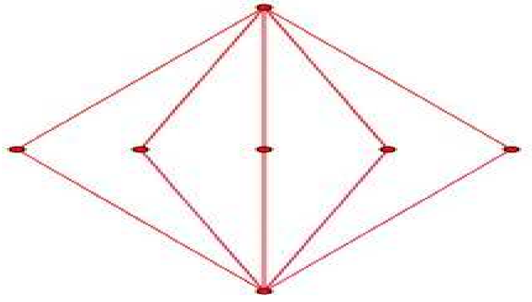


Figure 6.1: Input graph G

Set labelling method to construct the total graph of the given graph:

1. Using $|V|$ elements assign unique singleton label to each vertex of the input graph G.
2. Consider $|E|$ additional vertices. These vertices represent edges of G. For each new vertex, label it using a unique 2-element set, whose elements correspond to the endpoints of the edge which is represented by the newly added vertex.
3. Draw additional edges between pairs of vertices having non-empty singleton intersection between their corresponding labels.

4. The generated graph is the total graph of the given input graph.

An input graph is shown in Figure 6.1 and the corresponding generated total graph is shown in Figure 6.2.

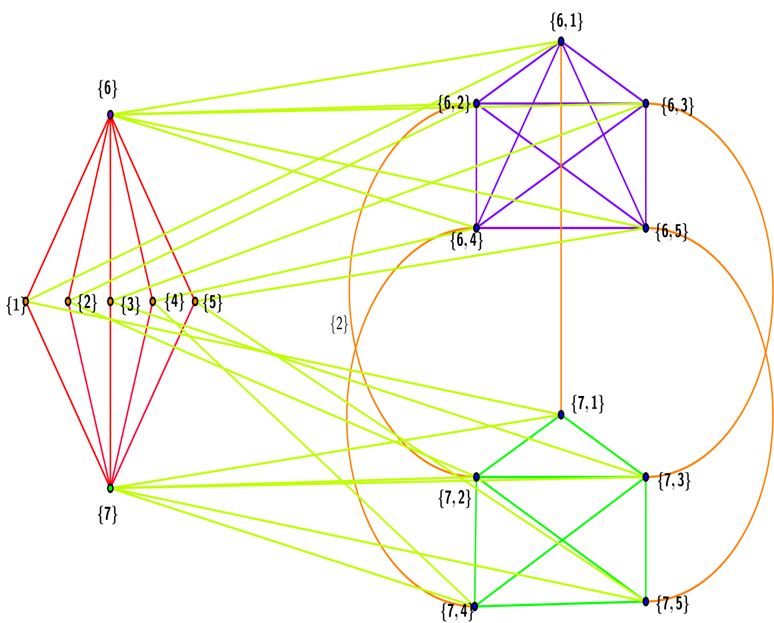


Figure 6.2: Total graph of G ($T(G)$)

6.2 Definitions and notation

In this section we present some of the basic definitions and notation we use in this chapter. All graphs we consider are finite, simple, undirected and connected. Disconnected graphs do not add any new dimension to the problem.

Definition 28. *The* **line graph** $L(G)$ *of a graph* $G = (V, E)$ *is defined as the graph with vertex set having one vertex corresponding to each edge in* G *and an edge between two vertices of* $L(G)$ *precisely when the edges of* G *that those vertices correspond to, have a common endpoint.*

Our main focus in this section is on total graphs. The vertex set of the total graph of a graph can be partitioned (uniquely upto isomorphism) into two parts such that the subgraph induced on one part is the original graph (inverse total graph) and the subgraph induced on the other is the line graph of the original

graph. The bipartite subgraph induced by this partition joins a vertex in the original graph to a vertex (representing an edge) in the line graph if and only if the vertex is an endpoint of the edge. Our interest in line graphs is limited to their role in total graphs, and their role in the partitioning procedure to obtain the inverse total graph.

Not all graphs are line graphs of a simple graph (or for that matter even of a multigraph). Similarly not all graphs are total graphs. Viewing the total graph concept as a function from the class of graphs to the class of graphs analogous to line graphs, the function is non-surjective. A natural problem, therefore, is to determine the range of this function. In addition, it is also interesting to design an algorithm that either reports that an input graph is not a total graph of any simple graph or returns the inverse total graph of the given total graph.

It has been established that the total graphs viewed as a function from the class of graphs to the class of graphs is injective [10].

Definition 29. *The total graph $T(G)$ of a graph $G = (V, E)$ has as vertex set one vertex for each edge as well as each vertex in G. Two vertices in $T(G)$ are adjacent precisely when the elements (vertex or edge) of G they represent are adjacent/incident to each other in G.*

With reference to a total graph $T(G)$ we have a partition of its vertex set into two parts, inducing G and $L(G)$ as explained in the previous paragraph. For a total graph such a partition is unique upto isomorphism and is called a **valid partition**. The individual vertices belonging to these two parts in a valid partition are introduced in the following definition.

Definition 30. *The vertex set of the total graph of a graph can be partitioned into:*

1. *The vertices of the original graph (we call such a vertex a* **vertex-vertex***)*

2. *The vertices of the line graph (we call such a vertex an* **edge-vertex***).*

Definition 31. *A* **mixed clique** *in a total graph is a clique which has at least one vertex from the set of vertex-vertices and at least one vertex from the set of edge-vertices in a valid partition of the total graph.*

Definition 32. *A* **pure clique** *in a total graph is a clique consisting exclusively of vertex-vertices or exclusively of edge-vertices.*

6.3 Total graphs: Basic properties

Theorem 6.3.1. *The largest mixed clique consisting of at least two vertex-vertices in a total graph is of size 3.*

Proof. Consider a clique consisting of three vertex-vertices. Clearly there is no edge in any graph incident to three distinct vertices. Hence this clique cannot be augmented to include any edge-vertices, and thus is not the subset of any mixed clique.

It follows that a mixed clique can consist of at most two adjacent vertex-vertices. In this case this can be augmented by only one vertex, the edge-vertex representing the link between these two adjacent vertex-vertices. □

Thus a maximal mixed clique is either:

- A vertex-vertex and all its adjacent edge-vertices; or
- Two adjacent vertex-vertices and the connecting edge-vertex.

In summary, a maximal mixed clique is either of size 3 or of size $k+1$ where $2k$ is the degree of its only vertex-vertex in the total graph. In fact, every edge of the original graph (inverse total graph) gives rise to a unique maximal mixed clique of size 3 involving one edge-vertex and two vertex-vertices. This is together with its two end points in the original graph. Such a triangle has two of its edges in the bipartite subgraph of the total graph induced by a valid partition and the third edge is between the two vertices in the vertex part.

Unique partitioning: To visualise this, view any graph as a union of stars centred at each of its vertices. These translate into complete graphs with number of vertices equal to the degree of the central vertex of the corresponding star, and lie in the line graph. Connect the central vertex of each star to each vertex of its corresponding clique in the line graph. Since we used a decomposition rather

than a partition, some pairs of vertices from these cliques in the line graph will have non-empty intersetion with respect to their neighbours in the original graph. Collapse them into identical vertices.

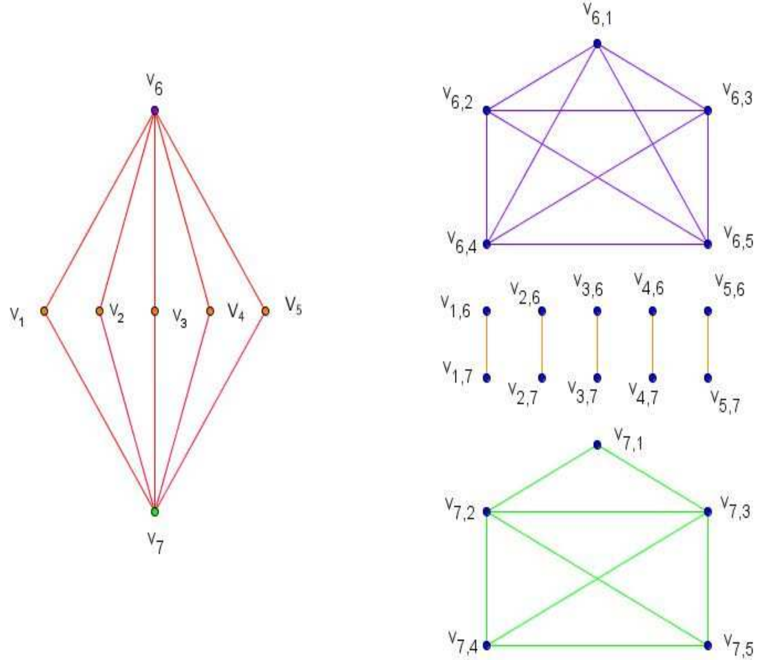


Figure 6.3: Line graph construction from the given graph

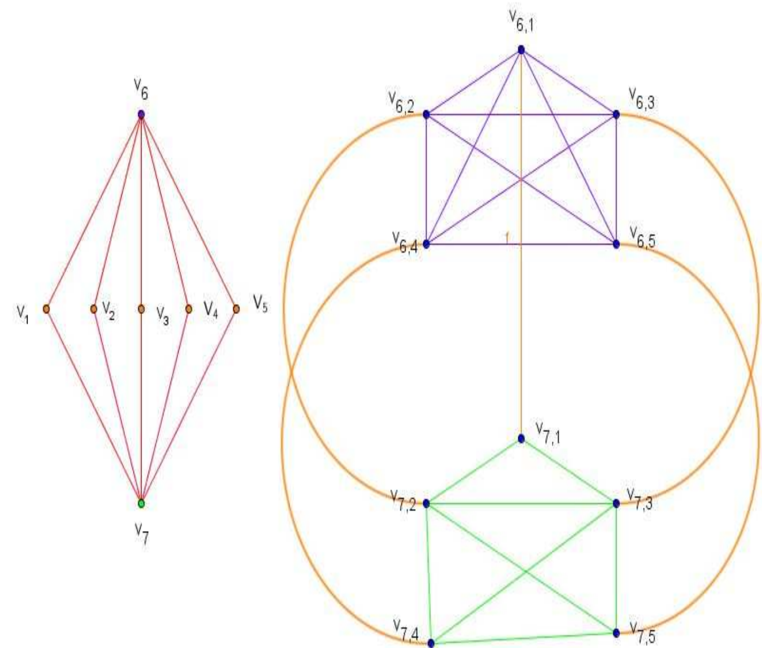


Figure 6.4: Line graph

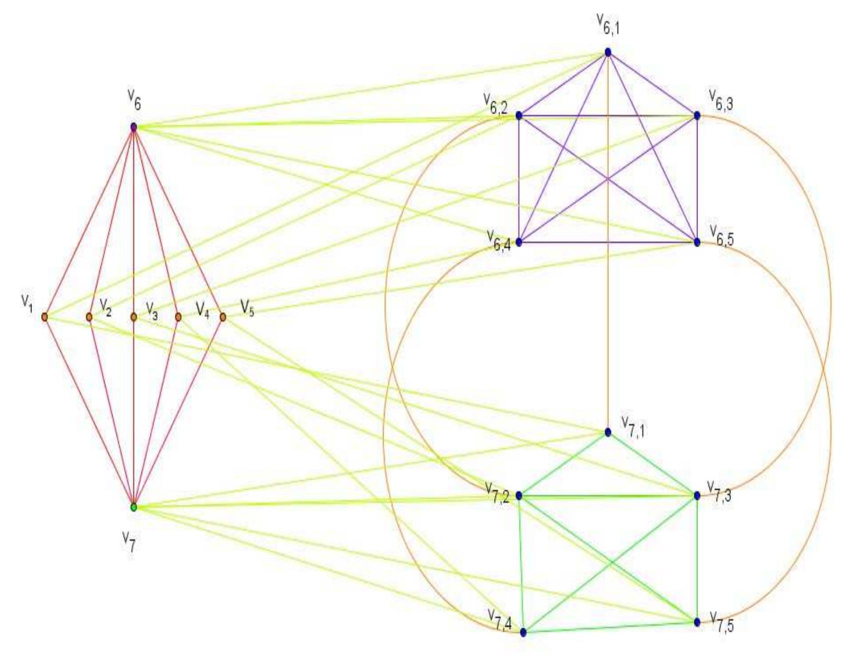


Figure 6.5: Total graph

6.3.1 Vertex degrees of $H = T(G)$ in terms of vertex degrees of G.

In this subsection, we derive results expressing the degrees of vertices of a total graph in terms of the degrees of vertices of its inverse total graph.

Theorem 6.3.2. *The degree of a vertex-vertex in a total graph is* 2 *times the degree of the original vertex in the inverse total graph.*

Proof. In the total graph a vertex-vertex is adjacent to vertices corresponding to its original neighbours as well as its incident edges in the original graph. See Figure 6.6. □

Theorem 6.3.3. *The degree of an edge-vertex in a total graph is equal to the sum of the degrees of the endpoint vertices of the original edge in the inverse total graph.*

Proof. In the total graph an edge-vertex is adjacent to its two endpoints (which are both vertex-vertices), and the other edges incident to these endpoints (which are all edge-vertices). Thus if its endpoints are u and v, its degree is $2 + (d_u - 1) + (d_v - 1) = d_u + d_v$. See Figure 6.7. □

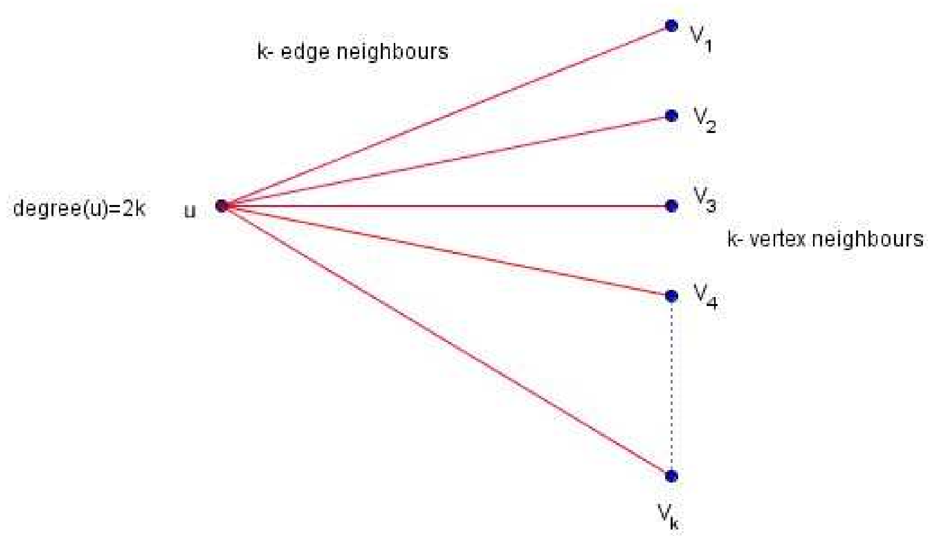


Figure 6.6: Degree characteristics of vertex-vertex

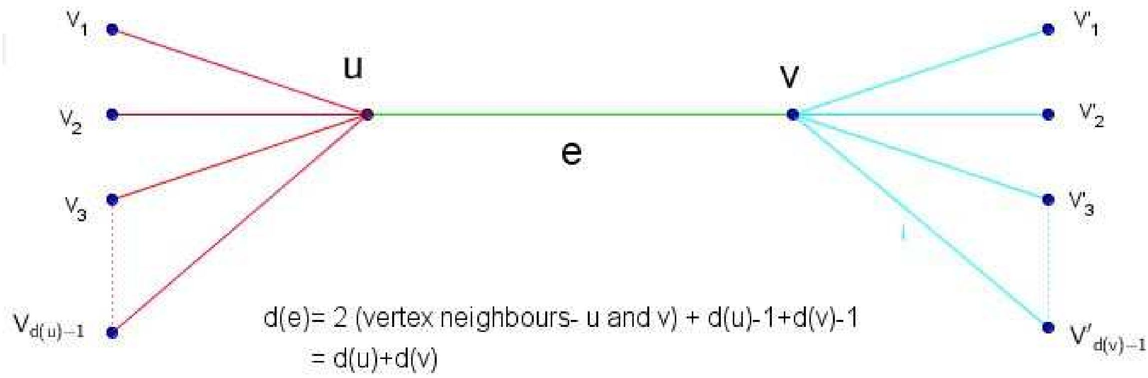


Figure 6.7: Degree characteristics of edge-vertex

The following theorem is an immediate consequence of Theorems 6.3.2 and 6.3.3.

Theorem 6.3.4. *The total graph of a graph is regular if and only if the original graph is regular.*

The next result follows from Definition 31 and Theorem 6.3.1.

Theorem 6.3.5. *Consider a maximal mixed clique of size 3 in a total graph. Let the two*

vertex-vertices be of degrees a and b, with $a > b$. Then the degree of the edge-vertex of this clique is $c = \frac{a+b}{2}$. Clearly $a > c > b$.

One can also immediately infer the following theorem.

Theorem 6.3.6. *In the inverse total graph there are two adjacent vertices of maximum degree if and only if there is a maximal mixed clique of exactly three maximum degree vertices two from the vertex part and one from the line graph part in the total graph.*

We will use this property extensively in our algorithm. If there is no triangle of vertices of maximum degree, a maximum degree vertex along with two of its neighbours with degree in arithmetic progression constitutes a mixed triangle with the highest and lowest degree vertices among them being from the vertex part and the mean value degree vertex from the edge part. This fact significantly reduces the work involved in finding a partition of the vertex set of the total graph into the vertex part and the line graph part iteratively.

Theorem 6.3.7. *Given a vertex-vertex v_a of degree $2k$ in a total graph, its neighbours can be divided into k pairs representing:*

- *Its distinct incident edges $(v_1, \ldots, v_k)$ in the inverse total graph and*
- *the other endpoints of those edges $(v'_1, \ldots, v'_k)$ respectively.*

The k pairs are $\{(v_1, v'_1), \ldots, (v_k, v'_k)\}$.

Proof. See Figure 6.10. The degrees of each pair is related to the degree of the selected vertex-vertex according to Theorem 6.3.5. □

6.3.2 Relationship between the number of vertices and edges of the original graph and its total graph

Here we extend the work of the previous subsection in a natural way obtaining results expressing the number of vertices and edges of a total graph in terms of the corresponding parameters of the inverse total graph.

Theorem 6.3.8. *Let $T(G) = (V', E')$ for the given $G = (V, E)$.*

1. $|V'| = |V| + |E|$

2. $|E'| \leq |E|(|V| + 1)$

Proof. The total graph of a given graph contains the original graph and the line graph of the original graph as disjoint induced subgraphs spanning all its vertices. The line graph of the original graph G has $|E|$ vertices. Therefore, $|V'| = |V| + |E|$

The edge set of the total graph can be partitioned into 3 disjoint sets:

A: Edge set of the original graph containg $|E|$ edges.

B: Edges which are present between the original graph and its line graph. There are $2|E|$ such edges.

Note: The line graph of the given G has $|E|$ vertices and each vertex of the line graph is connected to exactly 2 vertices of the original graph.

C: Edges which are present within the line graph. The number of such edges is

$$\begin{aligned}
\Sigma_{v\in V(G)} \binom{d_G(v)}{2} &= \Sigma_{v\in V(G)} \frac{d_G(v)(d_G(v)-1)}{2} \\
&= \left(\Sigma_{v\in V(G)} \frac{(d_G(v))^2}{2} \right) - |E| \\
&\leq \left(\Sigma_{v\in V(G)} \frac{(|V|-1)(d_G(v))}{2} \right) - |E| \\
&\leq (|V|-1)|E| - |E|
\end{aligned}$$

Adding the two equations and the inequality for the number of edges in groups A, B, C we get the claimed result. □

6.4 Results on complete graphs

We present an elegant direct construction method for the total graphs of complete graphs.

The following theorems give the structure and other parameters of the total graphs of complete graphs. Theorem 6.4.1 is an immediate consequence of Theorem 6.3.8.

Theorem 6.4.1. *Let $T(G) = (V', E')$ for the given $K_{|V|}$.*

1. $|V'| = \frac{|V||(|V|+1)}{2}$
2. $|E'| = \frac{|V|(|V|-1)(|V|+1)}{2}$
3. $\forall v \in T(G), d_{T(G)}(v) = 2(|V| - 1)$.

We now establish isomorphism between the line graph of a complete graph and the total graph of a complete graph with one fewer vertex.

Theorem 6.4.2. *$L(K_n) = T(K_{n-1})$ where $L(K_n)$ denotes the line graph of K_n.*

Proof.

- The line graph of K_n has exactly n cliques each of size $n-1$. These are formed by the stars induced by the edges incident to each of the n vertices of K_n. Each pair of these cliques share a unique common vertex. For instance the first clique has a distinct vertex common with each of the remaining $n-1$ cliques. The same holds for each of these cliques.
- $T(K_{n-1})$ contains $L(K_{n-1})$. It follows that there are exactly $n-1$ maximal cliques each of size $n-2$ in its induced line graph.
- $T(K_{n-1})$ also contains a copy of K_{n-1}, ignoring the vertices of the line graph portion. This is basically the inverse total graph.
- Thus, in total n cliques are present, $n-1$ of which are of size $n-2$ and the remaining 1 of size $n-1$. This follows from the previous two points.
- Each clique of size $n-2$ present in the line graph is adjacent to exactly 1 vertex of the original graph because the clique is formed by the vertices corresponding to edges incident on the single vertex of the original graph.
- Hence it is possible to include exactly 1 more vertex in each of the $n-1$ cliques of size $n-2$ present in the line graph.

- We conclude that a total of n cliques each of size $n-1$ are present in $T(K_{n-1})$. This is the same as in $L(K_n)$ including the overlapping pattern among these cliques.

□

Algorithm 2: Is a given graph the total graph of K_n

1. From Theorem 6.4.2, $T(K_n) = L(K_{n+1})$. For the given input graph, obtain its inverse line graph in $O(E')$ [42].
2. If the obtained inverse line graph is K_{n+1} then the G' is $T(K_n)$ (complexity of this step $O(E')$).

6.4.1 Direct method for construction of total graph of a complete graph ($T(K_n)$)

1. Consider $n+1$ disjoint groups each consisting of exactly n vertices. (These correspond to the $n+1$ cliques of size n of the total graph of the K_n).
2. The vertices in i^{th} group G_i are labelled $\{1, \ldots, n+1\} \setminus \{i\}$. Therefore, $|G_i| = n$
3. For each G_i, construct K_n by connecting all its n vertices pairwise.
4. Combine the jth vertex of group i and the ith vertex of group j into a single vertex. The neighbourhood of the new vertex is the union of the individual neighbourhoods. The degree of each new vertex is exactly $2(n-1)$ because degree of each original vertex is $n-1$.
5. The resultant graph is the total graph of the complete graph K_n.

Notes:

- No edge is destroyed during the entire procedure.
- Each clique has $\binom{n}{2}$ edges and initially $n+1$ distinct cliques are considered.

- Thus, $\binom{n}{2}(n+1)$ edges are present in the graph which remains constant through the course of this construction.

Construction of the total graph of K_3 using this direct method is shown in Figures 6.8 and 6.9. In the figure the group number is written as subscript for each vertex and the vertex number within the group is written in normal font.

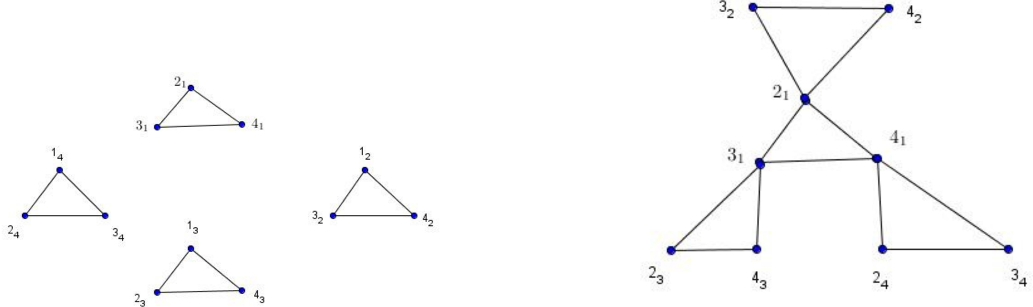


Figure 6.8: Initial steps of direct construction of the total graph of K_3

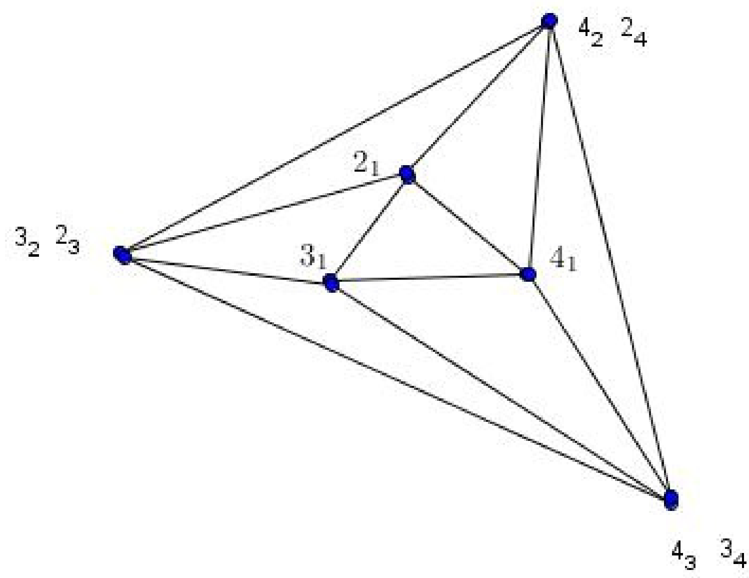


Figure 6.9: Resultant Total Graph of K_3 after step 4

6.5 Characterisation of total graphs and computing the inverse total graph

In this section, we prove two theorems which give conditions for a maximum degree vertex in a candidate total graph to be a vertex-vertex or an edge-vertex. These theorems are used to iteratively find a maximum degree vertex-vertex (one is guaranteed to exist by theorem 6.3.5) and partition its neighbours into vertex-vertices and edge-vertices. At each round a maximum degree vertex-vertex is selected as a part of the inverse total graph and it along with its edge-vertex neighbours are eliminated to get a smaller graph to recurse on. A partition of the vertex set of the given graph into the inverse total graph and the line graph of the inverse total graph is created iteratively, if one exists; or we infer that no such partition exists if there is a violation of the combinatorial conditions at any iteration. Our theorems in this section work only for total graphs of non-complete graphs. Thus the algorithm developed in this section also uses the algorithm of Section 6.4, when appropriate, to handle the case of complete graphs.

Theorem 6.5.1. *Given an arbitrary maximum degree vertex v, of degree $2k$, in a total graph $H = T(G)$, (G is not a complete graph) it is a vertex-vertex, if and only if the following properties hold in the subgraph induced on its open neighbourhood.*

1. *Its neighbours can be divided into two disjoint and exhaustive groups of k vertices each, one corresponding to its vertex neighbours $N^{\mathcal{V}}(v)$ in the inverse total graph and the other corresponding to its incident edges $N^{\mathcal{E}}(v)$ in the inverse total graph.*

2. *The maximum degree of $G[N^{\mathcal{V}}(v) \cup N^{\mathcal{E}}(v)]$ is k. This number is achieved by each vertex in $N^{\mathcal{E}}(v)$ and at least one vertex of $N^{\mathcal{V}}(v)$ falls short of this degree.*

Proof. The statements follow from an inspection of Figure 6.10. □

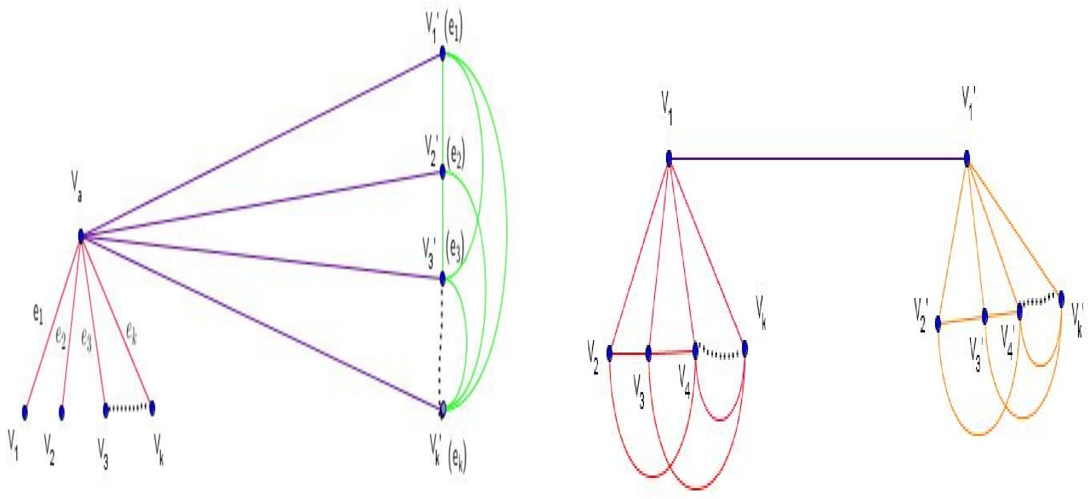


Figure 6.10: Characteristics of Vertex-Vertex and Edge-Vertex

Theorem 6.5.2. *Given an arbitrary maximum degree vertex v, of degree $2k$, in a total graph $H = T(G)$, (G is not a complete graph) it is an edge-vertex, if and only if the following property holds in the subgraph induced on its open neighbourhood.*

1. *Its neighbours can be partitioned into 2 maximal cliques of exactly k vertices each. Each of these cliques consist of one vertex-vertex and $k-1$ edge-vertices.*

Proof. The statement follows from an inspection of Figure 6.10. □

In 4-regular candidate total graphs (total graphs of cycles), each vertex satisfies the conditions of both the above theorems. That is because the two parts of any valid partition are isomorphic to each other.

Theorem 6.5.3. *Given any input (candidate) total graph $H = T(G)$, where G is not a complete graph, it is indeed a total graph if and only if the graph $H' = T(G')$ is a total graph where H' is obtained from G by eliminating a vertex-vertex of maximum degree along with all its edge neighbours in H.*

If the given graph is indeed a total graph, then it has a partition of its vertex set into vertex-vertices and edge-vertices. We find a maximal triangle of maximum degree vertices if one exists (Theorem 6.3.6). From this triangle, we obtain one vertex-vertex using the algorithmic version of Theorem 6.5.1.

If there is no triangle of maximum degree vertices, then it is guaranteed that any vertex of maximum degree, say v_a, is a vertex-vertex by Theorem 6.3.6. In this case each vertex of highest degree among its neighbours is an edge-vertex. Take one say v_c. By Theorem 6.3.5 there must be a common neighbour of v_a and v_c, say v_b such that the degree of v_c is the arithmetic mean of the degrees of v_a and v_b. Find one among these common neighbours and verify that it is indeed a vertex-vertex using Theorem 6.5.1.

Having found a candidate maximum degree vertex-vertex through one of the above two cases, verify if it is indeed a vertex-vertex using the algorithmic version of Theorem 6.5.1. If it is not a vertex-vertex then conclude that the input graph H is not a total graph. If it satisfies the conditions of Theorem 6.5.1 then partition its neighbours into vertex-vertices and edge-vertices. Deleting the vertex along with its edge neighbours, effectively eliminates the vertex and the incident edges from the inverse total graph (if it turns out to be a total graph). We are left with the (candidate) total graph of the graph with one vertex deleted. Hence by recursing on the smaller graph, we can obtain the partition or conclude that one does not exist if at some iteration there is a maximum degree vertex violating both Theorem 6.5.1 and Theorem 6.5.2. We thus have an algorithm which starts with a candidate total graph of a non-complete graph and decides whether it is indeed a total graph by recursing on the smaller graph or recourse to the complete graph theorem.

Algorithm 3: Inverse Total Graph.

1. **Check if** the given graph is the **total graph of a complete graph** using Algorithm 2. If so augument the vertices of that complete subgraph to the vertices obtained in earlier iterations and return.

2. Else consider maximal triangles containing three vertices of maximum degree using Theorem 6.3.6. If a such a triangle exists then one of its vertices must be a maximum degree vertex-vertex. Identify such a vertex x using Theorem 6.5.1. In the process of examining these 3 vertices, if any of them violates both Theorems 6.5.1 and 6.5.2 or no such x exists, conclude that the

input graph is not a total graph.

3. If no triangle of maximum degree vertices were found in step 2 then find a vertex x of maximum degree involved in a triangle as per Theorem 6.3.5. Check that x satisfies Theorem 6.5.1 and if not, conclude that the input graph is not a total graph.

4. If such an x was found in step 2 or 3, then partition its neighbours into vertex-vertices and edge-vertices by the algorithmic version of Theorem 6.5.1.

5. Add x to the vertex set of the inverse total graph and repeat the steps with the graph obtained by deleting x and its edge neighbours.

6. Return the set of vertices (and the induced subgraph on them) accumulated in Step 5 over all the iterations.

Runtime analysis of Algorithm 2:
Assume input graph is $G'(V', E')$ and the representation of G' is adjacency list representation.

Pre-processing:
Sort all the vertices in decreasing order of their degrees in $O(|V'|log|V'|)$.

1. Step 1: Equivalent to complexity of Algorithm 1: $O(|E'|)$

2. Step 2: Check for Theorem 6.3.6, $O(1)$. Theorem 6.5.1, $O(2k)$ for identification of neighbours and $O(|E'|)$ for checking condition 2 of Theorem 6.5.1. Same amount of time is required for checking conditions of Theorem 6.5.2. At most 3 times check for Theorems 6.5.1 and 6.5.2. Total complexity of this step : $O(|E'|)$.

3. Step 3: Check for Theorem 6.3.5, $O(|V'|)$. Theorem 6.5.1: $O(2k)$ for identification of neighbours and $O(|E'|)$ for checking condition 2.

4. Step 4: Partition: $O(|E'|)$

5. Step 5: Repeat steps 2, 3, 4 and 5 for $O(|V'|)$ times. i.e. $O(|V'| * |E'|)$

6. Step 6: $O(|V'|)$ time.

The complexity of Algorithm 3 is: $O(|V'| * |E'|)$ since $O(|V'| * |E'|)$ is the dominating term in $O(|V'|log|V'|)$+ $O(|E'|)$+ $O(|E'|)$ + $O(1)$+$O(|E'|)$+$O(|V'|)$+$O(|E'|)$ + $O(|V'| * |E'|)$+ $O(|V'|)$.

6.6 Conclusions & future directions

The work presented in this chapter has been published by us. The reference is [50].

We have proved properties of the vertex degrees of total graphs. We have developed a precise characterisation of the structure of the neighbourhoods of maximum degree vertices of the total graph of any graph. Combining these results we have designed an efficient iterative algorithm to compute the inverse total graph of a candidate total graph, or report that the graph is not a total graph. We also present a direct construction for the total graphs of complete graphs. One interesting direction of future research is to see if a given n and m pair admits a connected unique total graph if any. One can also look at minimum number of dynamic graph operations (adding/deleting vertices and edges or moving edges around the graph) to transform a non-total graph into a total graph.

Chapter 7

Spanning Tree Auxiliary Graph

In this chapter, we define a class of auxiliary graphs associated with simple undirected graphs. This class of auxiliary graphs is based on the set of spanning trees of the original graph and the edges constituting those spanning trees. A class of auxiliary graphs can be viewed as a function from the class of graphs to the class of graphs. We provide mathematical characterisation of graphs which are the spanning tree auxiliary graphs of some simple graph. Since the class of spanning tree auxiliary graphs of graphs do not have unique preimages (the forward function is not injective), we derive precisely the classes of graphs which have the same auxiliary graph. We design algorithms for computing a basic preimage and define rules to get other solutions for the same auxiliary graph. We also obtain several results expressing parameters of the auxiliary graph in terms of (not necessarily the same) parameters of the original graph.

Background details related to spanning tree auxiliary graphs are presented in Section 7.1. In Section 7.2 we present definitions and concepts used. We derive results on a few elementary standard graph parameters for the family of spanning tree auxiliary graphs in Section 7.3. In Section 7.4, we provide a classification of all maximal cliques that occur in the class of spanning tree auxiliary graphs. Section 7.5 discusses the role of prime graphs under the cartesian product operator as the building blocks of all spanning tree auxiliary graphs. The spanning tree auxiliary graphs of all 2-connected graphs are shown to belong to the family of prime graphs under the cartesian product operator. In Section 7.6 we provide an algorithm that recognises a graph that is a spanning tree auxiliary graph of a simple graph and computes a basic preimage, from which all preimages may be

generated. We summarise the results and possible future directions of research in Section 7.7.

7.1 Introduction

We study a class of auxiliary graphs where the vertices of the auxiliary graph represent the spanning trees of a given graph. There is an edge connecting two vertices of the auxiliary graph precisely when the symmetric difference of the edge sets of the corresponding spanning trees has exactly two edges. That is equivalent to saying that the two spanning trees have $(n-2)$ of their $(n-1)$ edges common.

Diagrammatically, one can label the vertices of a simple graph and also its edges with distinct labels. Given such a labelling of a graph G, one can label the vertices of the spanning tree auxiliary graph $Aux(G)$, each with the list of $(n-1)$ edges of the spanning tree it represents. From the description above it should be clear that we put an edge between two vertices in the spanning tree auxiliary graph if and only if the labels of the two vertices share $(n-2)$ of their $(n-1)$ elements in common. See the figure below for a graph G and its spanning tree auxiliary graph $Aux(G)$.

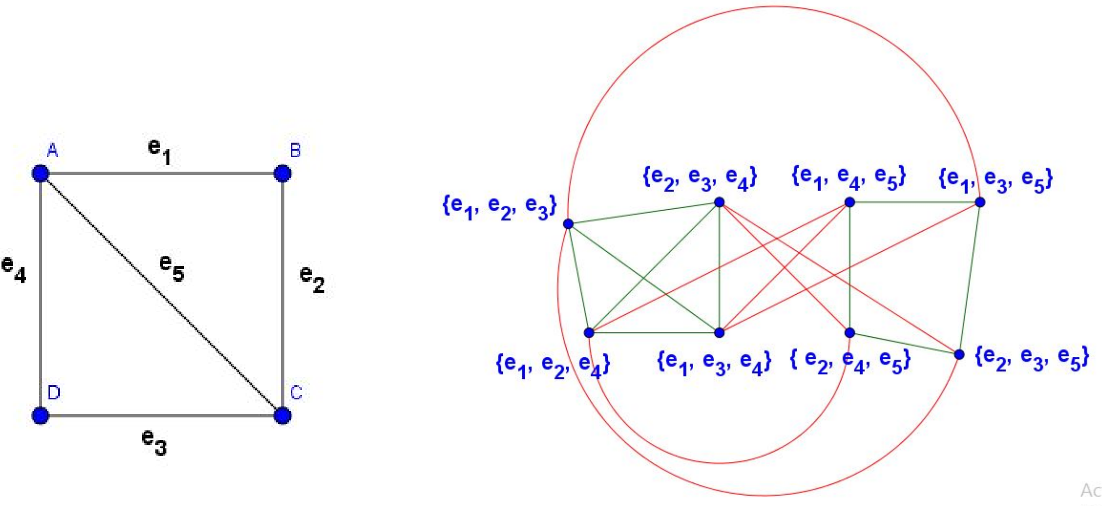


Figure 7.1: Construction of $Aux(G)$ from G

This way of migrating from one spanning tree of a graph to another has already been studied in various places and a similar notion forms the basis of the

proofs of correctness of algorithms such as Prim's and Kruskal's [40] for computing minimum spanning trees in weighted graphs. Counting or enumerating the spanning trees of graphs has been extensively studied in the literature [13] [35] [57]. This underlines the importance of this class of graphs. Apart from applications of this class of graphs in various problems as described here, it is also challenging combinatorially and algorithmically to characterise this family of graphs.

Here, we formalise the notion of spanning tree auxiliary graphs of graphs and characterise them in terms of their mathematical properties. We develop algorithms for recognising such graphs and computing an inverse solution. We provide a complete description of all graphs which constitute preimages on the basis of one basic preimage (the spanning tree auxiliary graph of graphs is not an injective function and each point in the range has infinitely many preimages). We derive relations between parameters of a given graph and (not necessarily the same) parameters of the corresponding auxiliary graph.

Throughout, we assume the original graph for which we are considering spanning tree auxiliary graphs is a simple undirected graph. We deal only with connected graphs because the set of disconnected graphs have no spanning trees and hence the corresponding spanning tree auxiliary graphs are trivial, with zero vertices.

7.2 Definitions

A tree is a simple undirected connected acyclic graph. A connected unicyclic graph is any graph obtained by augumenting a tree with an edge between a non-adjacent pair of vertices.

Given any spanning tree T of a simple connected undirected graph G, adding any edge $e \in E(G) \setminus E(T)$ results in a unicyclic graph U. Since, G is a simple graph, the unique cycle in U must necessarily be of length at least 3. Suppose it is of length k, then there are exactly $(k-1)$ non-cut edges in U different from e. Deleting any one of them results in a spanning tree T' of G, different from T. We call this process of adding an edge to a spanning tree of a connected graph and

deleting **some other** edge from the unique cycle thus introduced, a **unit transformation of type** 1. Any two spanning trees can be constructed from one another by a series of unit transformations of type 1, and in fact in at most $(n-1)$ unit transformations where n is the number of vertices of G.

Given any spanning tree T of a graph G, deleting any edge e from T results in a spanning forest of G consisting of exactly two trees T_1 and T_2. Adding any edge of the original graph different from e and linking a vertex of T_1 to a vertex of T_2 results in a spanning tree T' different from T. The number of such edges is equal to the number of edges in G between the vertex partition defined by the vertices of T_1 and T_2. We call this process of deleting an edge of a spanning tree and relinking the two resulting subtrees by a **different** edge a **unit transformation of type** 2.

Any two spanning trees can also be constructed from one another by a series of unit transformations of type 2, and in fact in at most $(n-1)$ unit transformations where n is the number of vertices of G. It should also be evident that any two spanning trees can be constructed from one another by a mixed series of type 1 and type 2 unit transformations, again requiring no more than $(n-1)$ steps in the most efficient way.

Definition 33. *Given a simple graph G, we define its* **spanning tree auxiliary graph** *$Aux(G)$ as the graph which has a vertex corresponding to each spanning tree of G, and two vertices of $Aux(G)$ are adjacent if and only if the corresponding spanning trees in G can be obtained by a single unit transformation.*

Definition 34. *A graph G is* **2-connected** *if it cannot be disconnected by deleting fewer than two vertices. In particular, the graph itself must be connected, because otherwise, it is rendered disconnected by removing zero vertices, which is fewer than two.*

Definition 35. *The* **circumference** *of a graph is the length of any longest cycle in a graph.*

Definition 36. *An* **edge cut** *is an edge set of the form $[S, \overline{S}]$, where S is a non-empty proper subset of $V(G)$ and $\overline{S}$ denotes $V(G) - S$.*

Definition 37. *A* **minimal edge cut** *is an edge cut such that if any edge is put back in the graph, the graph will be reconnected.*

Definition 38. *A* **maximum-minimal-edgecut** *is a minimal edge cut of maximum size.*

Definition 39. *A* **block** *in a graph is defined as any maximal* 2*-connected subgraph of a graph.*

It is an elementary result that any two blocks in a graph can share at most one common vertex. A useful auxiliary graph to study the block structure of a connected graph is the standard **block-cutpoint tree** [25] of a graph. The block-cutpoint tree of a graph is computed by a standard algorithm which is an adaptation of depth first search (DFS).

We would like to state at the very outset that there are infinitely many graphs which all map to the same *Aux* graph, and hence we need to develop a notion of a canonical/minimal preimage.

Definition 40. *A* **minimal preimage** *of a spanning tree auxiliary graph is a connected graph none of whose blocks is K_2. The blocks in such a listing maybe linked together in any form allowed by the standard block-cutpoint tree concept.*

The motivation behind the above definition is that the only changes to a graph that do not alter the spanning tree auxiliary graph are addition of blocks which are all K_2.

We now define the notion of **ear addition** as used by us. The concept is not a new one, but our definition is slightly different and hence we present it here.

Definition 41. *We define an* **ear addition** *as an extension of a graph by adding a path through zero or more new vertices with two distinct existing vertices of the graph as the endpoints of the path.*

If the endpoints of the path are already adjacent then the ear must contain at least one intermediate vertex since we consider only simple graphs. An **ear decomposition** of a graph is the reconstruction of the graph from scratch by first drawing one of its cycles and then repeatedly adding an ear.

We now state Whitney's Theorem [62] on 2-connected graphs..

Theorem 7.2.1 (Whitney's Theorem). *A graph is* 2*-connected (apart from* K_2*) if and only if it can be obtained by starting with a cycle and performing zero or more operations of ear addition.*

Observation 1. *In* 2*-connected graphs, every pair of edges has at least one cycle containing both.*

As a subsidiary goal this section develops relations between graph parameters of $Aux(G)$ and (not necessarily the same) parameters of G. This intermediate step helps towards the main goal and is also interesting in its own right.

Definition 42. *Given two graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$*, the cartesian product* $H = G_1 \square G_2$ *has vertex set* $V = V_1 \times V_2$ *where* $\times$ *represents the cartesian product of the two vertex sets and an edge connects* (u_1, u_2) *to* (v_1, v_2) *if and only if* $u_1 = v_1$ *and* $(u_2, v_2) \in E_2$ *or* $(u_1, v_1) \in E_1$ *and* $u_2 = v_2$*.*

This operator defined for two graphs can be extended iteratively to any number of graphs. The operation is commutative and associative in the sense that the graphs obtained by commuting or bracketing a series of graphs in any order gives rise to the same product graph upto isomorphism. The graph K_1 serves as the identity for the cartesian product operator on graphs. It is well known that the nontrivial factors of a graph under the cartesian product operator are unique upto reordering.

Definition 43. *A graph which has no nontrivial factors under the cartesian product operator is called prime.*

A graph obtained as the cartesian product of k nontrivial factors [29] [30] is a graph of dimension k under the cartesian product operation. Each vertex in the product graph involving k nontrivial factors is a k dimensional vector where the ith coordinate is a vertex from the ith factor in the product. The degree of a vertex in a graph obtained as the cartesian product of several graphs is the sum of the degrees of the vertices in each of its coordinates in the corresponding factor graphs.

Observation 2. *Since nontrivial factors involve a minimum degree of at least 1, the presence of every edge of a vertex in the same factor implies the graph is prime.*

The dimension of each vertex is identical and is the same as the dimension of the graph under cartesian product. Thus, in order to establish that a graph is prime under the cartesian product operator, it is enough to establish that all edges incident to some vertex belong to the same factor. It is also known that all edges of a clique of size three or more in a cartesian product of graphs must all come from the same factor.

7.3 Parameters

In this section we give some elementary results on some standard graph parameters of spanning tree auxiliary graphs of graphs.

Lemma 7.3.1. Max. degree:

$$\Delta(Aux(G)) \leq (n(G) - 1) * (m(G) - n(G) + 1)$$

Proof. For a vertex of $Aux(G)$ there is an associated spanning tree, the number of edges of G not belonging to it is $m(g) - n(G) + 1$. For each of those edges, adding them to the tree results in a cycle. The length of this cycle is at most $n(G)$, and thus the number of edges on the cycle, different from the one that was added is at most $n(G) - 1$. Removing any of these edges generates a new spanning tree of G and thus a neighbour of the vertex considered in $Aux(G)$. Combining these observations gives the upper bound on the maximum degree. □

Lemma 7.3.2. Min Degree:

$$\delta(Aux(G)) \geq 2 * ((m(G) - n(G) + 1)$$

Proof. The proof is almost identical to the previous lemma, the only difference is that we use the lower bound on the length of the cycle created, rather than the upper bound. The lower bound is 3, since we are dealing with simple graphs. The rest of the arguments are identical. □

Lemma 7.3.3. *The diameter* $diam(Aux(G)) \leq n(G) - 1$.

Proof. The minimum number of operations to transform one spanning tree to the other is the size of the set difference of the edge sets of the two spanning trees. This can never be more than the number of edges in the tree, this bound being achieved in case of edge disjoint spaning trrees. Thus the result follows. □

Lemma 7.3.4. *The* **Clique Number:**
$\omega(Aux(G)) = Max\{Circumference(G), |maximum_minimal_edgecut(G)|\}$

Proof. Every maximal clique in $Aux(G)$ corresponds to either a cycle in G or a minimal edge cut in G as explained along with the definitions of the two types of unit transformations. Thus the maximum clique size in $Aux(G)$, which is necessarily a maximal clique is a largest among these. Thus the result follows. □

7.4 Classification of maximal cliques in $Aux(G)$ in terms of structures in G

Here we describe cliques on three or more vertices in $Aux(G)$. Each spanning tree of a graph G has exactly $(n-1)$ edges where n is the number of vertices in G. Consider a clique of size three in $Aux(G)$. This clique represents three spanning trees of G each pair among which there is exactly $(n-2)$ common edges. There are two possibilities for the common intersection of the edge sets of all three spanning trees. Either it is $(n-3)$ or it is $(n-2)$. If we consider any fourth vertex to augment the three clique to a four clique, then in the first case, the common intersecton of the edge sets of the four spanning trees will go down to $(n-4)$, while in the second case it will remain $(n-2)$. The same logic extends to larger cliques. If it is a clique of type 1, then the common intersection decreases for each added vertex, while if it is of type 2, the common intersection remains $(n-2)$. Structurally cliques of type 1 arise from cycles in G and cliques of type 2 arise from minimal edge cuts in G.

The neighbourhood of each vertex in $Aux(G)$ can be partitioned into maximal cliques in these two different ways. In the first case the number of cliques in the

partition is $m - n + 1$ one corresponding to each edge of G not in the spanning tree T. In the second case the number of cliques in the partition is $n - 1$, one for each edge in the spanning tree T.

Each maximal clique in $Aux(G)$ is a direct and exclusive consequence of either a cycle in G or a minimal edge-cut in G. The size of the cliques resulting in these two cases are respectively the length of the cycle and the number of edges in the edge-cut respectively. To summarise:

Due to cycles:

Take any cycle C of length k in G. Consider a spanning tree T which uses some $(k-1)$ of the edges of this cycle. Let F be the forest resulting by deletion of these $(k-1)$ edges from T. Clearly appending any path of $(k-1)$ edges of the cycle C to the edges of F result in a spanning tree of G and differ from any other such tree in exactly one edge. Thus these are all pairwise adjacent and form a maximal clique of size k in $Aux(G)$.

Due to minimal edge-cuts:

We assume G is connected and let $\mathcal{E}$ constitute a minimal edge cut of G containing k edges. The deletion of the edges of $\mathcal{E}$ results in a two component graph. Take any fixed spanning forest of this two component graph containing spanning trees T_1 and T_2 of the two components respectively. Cross connecting T_1 and T_2 with any of the k edges of $\mathcal{E}$ results in a spanning tree of G. Clearly each of these spanning trees differ from each other in exactly one edge. Thus, they constitute a maximal clique of size k in $Aux(G)$.

There are two basic ways of creating a new spanning tree of a graph starting from a given spanning tree of the same graph. These are very similar to each other as single operations go but when we consider a series of these operations (or more precisely a large number of possibilities of completing the second phase of these operations) the difference between them becomes important and hence we consider both.

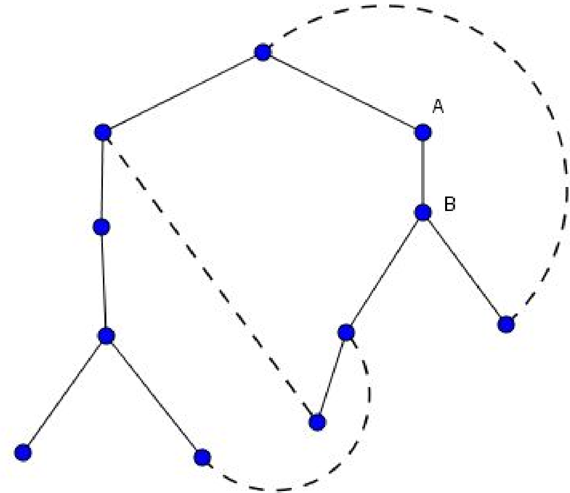


Figure 7.2: Idea of Type I and Type II cliques for edge (A.B)

The first is to add an edge of the original graph not present in the spanning tree creating a unicyclic graph and then deleting some edge different from the one that was added that belongs to the unique cycle in the unicyclic graph. This is what we called a unit transformation of Type I in our earlier definition.

The other method which is almost a dual of the previously mentioned one is to first delete an edge of the spanning tree and interconnecting the two subtrees thus formed using some different edge of the original graph that links a vertex from one subtree to a vertex of the other subtree. This is what we called a unit transformation of Type II in our earlier definition.

7.5 Minimal preimage and multiple preimages

Here we describe the properties which make two or more graphs map to the same Aux graph.

Theorem 7.5.1. *The auxiliary graph of G consisting of blocks $B_1, \ldots, B_k$ is the cartesian product of the individual auxiliary graphs. That is:*

$$Aux(G) = Aux(B_1) \square \cdots \square Aux(B_k)$$

Proof. Consider any graph G and any spanning tree T of G. Let B be some block

of G and let $T[B]$ be the subgraph of T induced by the vertices of B. Clearly $T[B]$ is a tree. If it were a forest with more than one component, it means there exists a path leaving the vertices of B and coming back linking distinct vertices of B via a path with vertices outside B. This contradicts the assumption that B is a block.

Thus, the spanning trees of any connected graph can all be be obtained by patching together in any way individual spanning trees of each block of G. In fact any spanning tree of G can be obtained by this procedure and and any tree resulting from this patching together of spanning trees of blocks is also a spanning tree of G. □

It follows from the above that any spanning tree of a graph can be viewed as an (ordered) list of spanning trees of its individual blocks. Different spanning trees of the graph can be obtained by starting with some spanning tree and then varying independently the spanning trees of each block. In other words the set of all spanning trees of the graph can be obtained as vectors of dimension k where k is the number of blocks of G.

We may also recall that in $Aux(G)$, two vertices (representing two distinct spanning trees in G) are adjacent if and only if they can be obtained from each other via a unit transformation. Also the edges involved in this unit transformation must both come from the same block of G since they form a part of a cycle in G and there can be no cycle crossing more than one block. Thus we can say that the two "adjacent" spanning trees agree in their restriction in all blocks except one, and on the one where they disagree, they differ by a unit transformation. If we were to treat these spanning trees a k dimension vectors one for each block of G, then it is like the cartesian product of the individual graphs.

Theorem 7.5.2. *If G' is obtained from a graph G by iteratively appending new blocks to G each of which is a K_2 results in no change in $Aux(G')$ from $Aux(G)$.*

This is because $Aux(K_2) = K_1$ and $Aux(G)\Box K_1 = Aux(G)$.

Theorem 7.5.3. *Let G be a connected graph with no cut edges, and let $H = Aux(G)$. Then H is a prime graph under the cartesian product operation if and only if G is a 2-connected graph.*

As already argued above, the spanning tree auxiliary graph of an arbitrary graph is the cartesian product of the spanning tree auxiliary graphs of each of its blocks. Thus it only remains to demonstrate the converse. Here we focus on an arbitrary vertex x in $Aux(G)$ and argue that all the edges incident to x in $Aux(G)$ come from the same factor. This will imply that $Aux(G)$ is prime under the cartesian product operator. We will, of course, have to use the fact that G is 2-connected in the course of our proof.

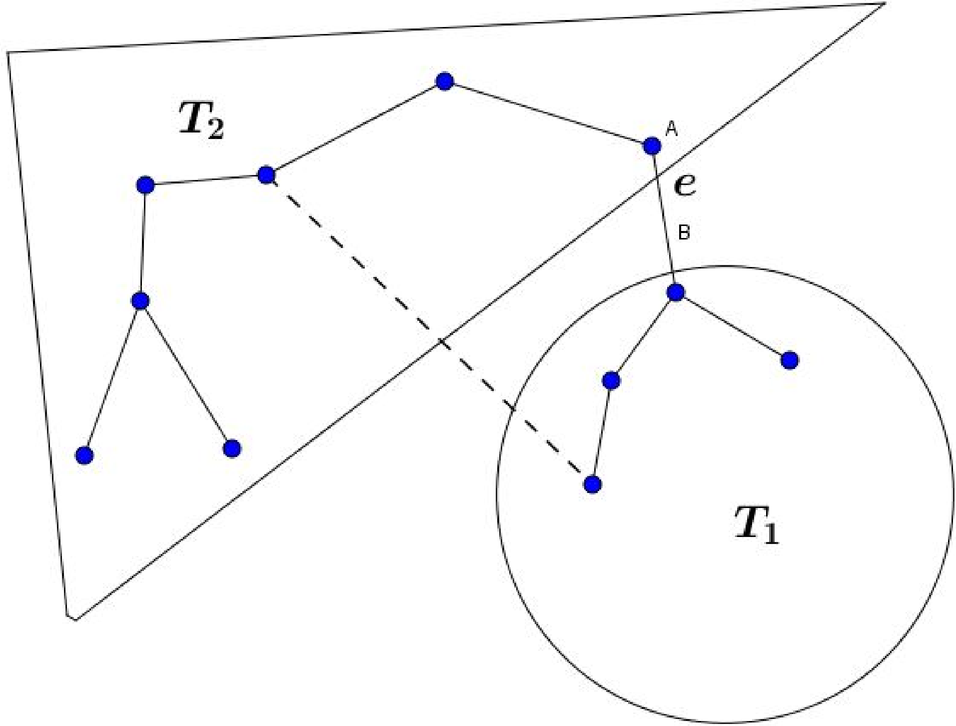


Figure 7.3: Deletion of the edge e is corresponding to the cycle clique in the $Aux(G)$

Consider the spanning tree T_x of G corresponding to the vertex x in $Aux(G)$. The edges incident to x in $Aux(G)$ connect it to its neighbours. Hence, these correspond to spanning trees of G obtained from T_x by a single unit transformation. Consider an edge e in T_x. Deleting e from T_x results in a spanning forest of G with exactly two trees T_1 and T_2. Since G is 2-connected, there is at least one edge in G apart from e linking the vertices of T_1 to the vertices of T_2. If there is exactly one such edge then it corresponds to a cycle clique in $Aux(G)$ and if there is more

than one then they together form a minimal edge cut clique in $Aux(G)$. Hence in both these cases all these edges incident to x are from the same factor.

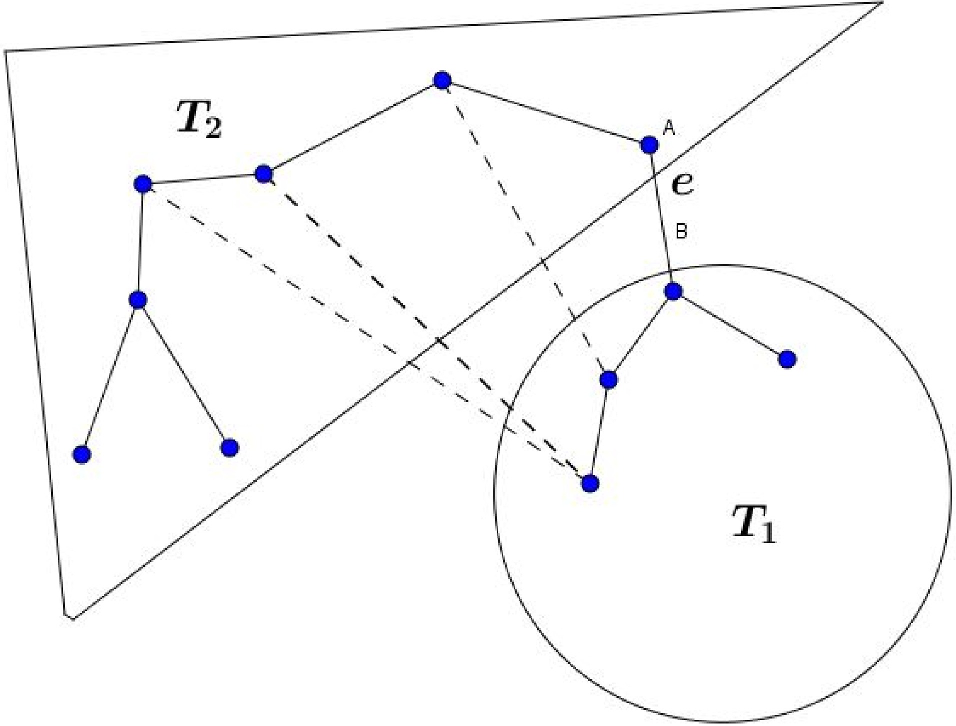


Figure 7.4: Deletion of the edge e is corresponding to the minimal edge cut clique in the $Aux(G)$

Now consider two incident edges e_1 and e_2 in T_x. One can consider constructing T_x from G by repeatedly deleting edges that lie on a cycle until the graph is acyclic. Fix all cycles of the graph containing both e_1 and e_2 (by observation 1 there is at least one such cycle). Remove an edge from every cycle except the cycles which contain both e_1 and e_2. now all remaining cycles have both these edges left, and hence the last edge deleted will destroy a cycle containing both e_1 and e_2. Let this deleted edge be e'. Thus there is a cycle clique in $Aux(G)$ corresponding to trees obtained by selecting all but one edge on this cycle and all other edges from T_x. This includes the node x and hence all these edges incident to x and involving unit transformation deleting incident edges in T_x also come from the same factor.

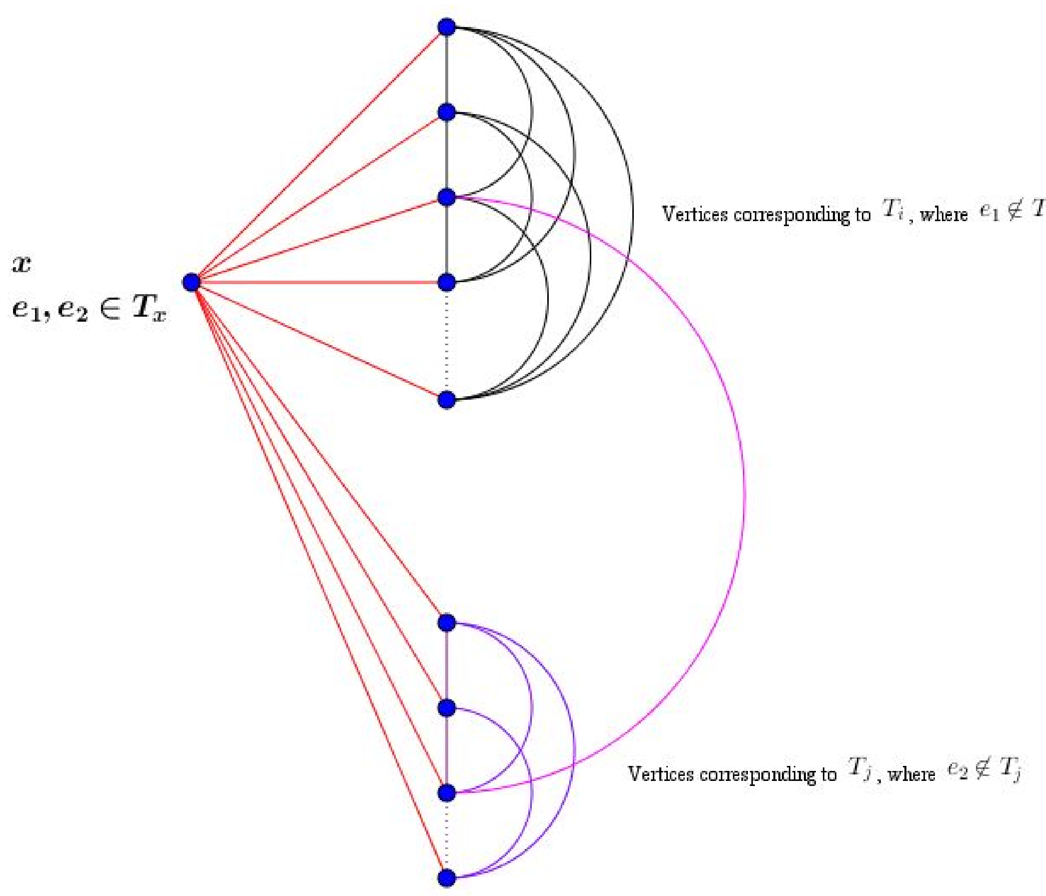


Figure 7.5: All edges incident to x are from the same factor

Now extending this idea for successive edges incident to each other on a path in T_x we have established that all edges incident to x are from the same factor. From observation 2, $Aux(G)$ is prime under cartesian product.

Thus $Aux(G)$ is prime if G is 2-connected and has no cut-edges.

Observation 3. *$Aux(T) = K_1$ for any tree T.*

This follows because a tree has exactly one spanning tree. Note that the spanning tree auxiliary graph of a tree- the complete graph on one vertex, K_1 - is also the identity for the cartesian product operator. Thus, from the previous two theorems, we conclude, that appending any number of blocks which are trees to a given graph, does not alter the spanning tree auxiliary graph of that graph. Thus minimal preimages contain no cut-edges.

7.6 Characterisation of Aux

Here we describe some properties of spanning tree auxiliary graphs and show how they can be applied to developing ideas for an algorithm to compute the inverse.

Type I clique: After the addition of the edge, deleting any edge of the cycle created constitutes a clique in the neighbourhood of the vertex corresponding to the original tree in the auxiliary graph. Thus corresponding to a cycle of length k in the original graph any spanning tree consisting of $(k-1)$ of these edges has in its neighbourhood a clique of size k associated with this cycle. The number of such cliques in the neighbourhood is equal to the number of edges in the graph not part of the spanning tree. This number is clearly equal to $m(G) - n(G) + 1$. This is therefore the number of such cliques into which the neighbourhood of each vertex can be partitioned.

Type II clique: After the deletion of the edge, adding eny edge bridging the resulting two subtrees results in a clique. The number of vertices in the neighbourhood of the vertex associated with this spanning tree in the auxiliary graph is equal to the number of other edges crossing this minimal edge cut. Thus it results in a clique of size $(k-1)$ in the neighbourhood of this vertex in the auxiliary graph. This kind of operation can be performed with any of the $(n-1)$ edges of the tree and hence the neighbourhood of a vertex in $Aux(G)$ can be partitioned in a different way into exactly $(n-1)$ maximal cliques.

Observation 4. *Every clique of size* 3 *or more in* $Aux(G)$ *uniquely extends to a maximal clique. Therefore these cliques can be computed in polynomial time.*

7.6.1 Algorithm for computing a basic preimage

Thus, one can perform the operation of partitioning the neighbourhod of each vertex in the auxiliary graph into cliques in two ways and obtain simultaneous linear equations in the two variables n and m. This results (surprisingly) in two possible solutions, because the right hand side numbers in these two equations can be exchanged. Any graph which has a nontrivial spanning tree auxiliary graph

must be connected since it has spanning trees. We use these along with properties developed in Section 7.3 to get values of m and n.

The simultaneous equations described above must be consistent across all vertices of a potential $Aux(G)$, in order for the graph to be a spanning tree auxiliary graph. If it fails for at least one vertex then the graph is not a spanning tree auxiliary graph of any graph.

Having obtained values of $m(G)$ and $n(G)$, we now focus on the subgraph of $Aux(G)$ induced by some arbitary vertex x and its neighbourhood $N(x)$. Call this subgraph H. H essentially reprents one spanning tree of G namely x and all other spanning trees obtainable from x by applying a single unit transformation. We do not have the actual structure of T or any of the other trees represented by vertices in H. We had earlier computed two partitions of the neighbourhood of each vertex of $Aux(G)$ into maximal cliques. One of the partitions represents cycle cliques and the other represents minimal edge-cut cliques. Having obtained the values of $m(G)$ and $n(G)$, we know which of the two partitions reprents cycle cliques and which represents minimal edge-cut cliques. Hence, with respect to the spanning tree of G associated with x, we can use sizes of the maximal cliques in the cycle clique partition to list the lengths of all fundamental cycles of G. Similarly, we use the edge-cut clique partition to list the number of edges of G connecting the vertex sets of the two subtrees obtained by deleting any edge of the spanning tree associated with x.

Theorem 7.6.1. *We have reduced the problem of computing* 2*-connected G from* $Aux(G)$ *to reconstructing a graph from the following information.*

- *Number of vertices* $n(G)$
- *Number of edges* $m(G)$
- *The lengths of all fundamental cycles with respect to a specific spanning tree, (say* T*).*
- *The number of edges of G crossing between the vertex sets of the two subtrees obtained by deleting any edge of* T*.*

- *Other results mentioned in Section 7.3.*

Given an arbitrary graph we first compute its prime factors under the cartesian product operation using well known algorithms [4]. The above result applies only to 2-connected instance of G (and consequently $Aux(G)$ is prime under cartesian product). For graphs which are not 2-connected the algorithm uses Theorem 7.5.3 to reduce into several subproblems and then apply the above result.

7.6.2 Analysis

We give here a brief informal analysis of running time of the algorithm provided by us.

- Prime factors of the input graph can be computed in polynomial time [15].
- From Observation 4, it is possible to compute all the maximal cliques in polynomial time (assuming the graph is the spanning tree auxiliary graph of some graph). This can be done in $O(n^4)$ time because there are $O(n^3)$ triangles and each extends greedily to unique maximal clique.
- If the decomposition of the previous step is consistent across all the vertices then $n(G)$ and $m(G)$ can be computed in polynomial time.
- The reduction indicated by Theorem 7.6.1 can be done in polynomial time.

We are currently working on constructing graphs from the information that results after applying the reduction of Theorem 7.6.1.

7.7 Conclusions

We have looked at the important class of spanning tree auxiliary graphs and given a mathematical characterisation of such graphs. We have also translated the mathematical result into efficient algorithmic ideas for recognising graphs of this class. This idea can often help in computing the inverse graph, but there is scope to improve the algorithmic idea into a concrete and precise algorithm.

Further studies of properties of this class of graphs and improved algorithms for recognition are possible future directions of related research particularly for an algorithmic version of Theorem 7.6.1.

CHAPTER 8

Conclusions

We have obtained optimal value of USN for the complement of complete graphs, complete graphs, complete bipartite graphs and upper bound for paths, cycles, matching, hypercube, wheel graph etc. Results on path, cycle, matching and wheel graphs are asymptotically tight for all three problem variants. In the future we plan to derive optimal and/or lower bound results for hypercube, harary graph etc and solve possible variants.

Table 8.1: Summary of results

	USN	UUSN	ILN
Lower Bound	$\lceil log_2 n \rceil$	$\lfloor log_2 n \rfloor + 1$	1
Upper Bound	$(n-1)+\binom{n}{2}$	$n+\binom{n}{2}\ (n-1)$	n
K_n	$n-1$	n	1
$\overline{K}_n$	$1+\lceil \log_2 n \rceil$	$1.5(1+\lceil \log_2 n \rceil)$	2
M_{2n}	$O(\log_2 2n)$	$O(\log_2 2n)$	$O(\log_2 2n)$
$K_{s,t}$	$2+\lceil \log_2 s \rceil + \lceil \log_2 t \rceil$	$1.5(2+\lceil \log_2 s \rceil + \lceil \log_2 t \rceil)+$ $\lvert \lceil \log_2 s \rceil - \lceil \log_2 t \rceil \rvert$	2
P_n	$O(logn)$	$O(logn)$	$O(logn)$
C_n	$O(logn)$	$O(logn)$	$O(logn)$
W_n	$O(logn)$	$O(logn)$	$O(logn)$
Q_n	$3n+O(logn)$	$3n+O(logn)$	$n+O(logn)$
BT_n	$O(logn)$	$O(logn)$	$O(logn)$

Our proposed interactive data visualization technique **(Labeled Object Treemap)**

displays multiple hierarchies without any information loss and it can display larges data sets coherently. Our proposed technique generates labelling of all objects automatically. The interactive version offers various features using which it is easy to identify specific types of neighbors of a node just by clicking on it. Taxonomy is visible using treemap. The proposed interactive version has all the characteristics (single representation, compactness, clusters identification etc) to be a good visualization technique for visualizing multiple hierarchies.In future, one can think of an arrangement of nodes that is more compact so that a larger number of nodes can be clearly shown (along with the labels) in the same available space. Perhaps, arranging the nodes in a zigzag pattern is one such solution.

We have shown how our proposed graph based information visualization technique **(Edgeless Graph)** can be used for the analysis of a social network and dynamic changes in the underlying social network are also considered.

We have proved properties of the vertex degrees of total graphs. We have developed a precise characterisation of the structure of the neighbourhoods of maximum degree vertices of the total graph of any graph. Combining these results we have designed an efficient iterative algorithm to compute the inverse total graph of a candidate total graph, or report that the graph is not a total graph. We also present a direct construction for the total graphs of complete graphs. One interesting direction of future research is to see if a given n and m pair admits a connected unique total graph if any. One can also look at minimum number of dynamic graph operations (adding/deleting vertices and edges or moving edges around the graph) to transform a non-total graph into a total graph.

We have looked at the important class of spanning tree auxiliary graphs and developed structural and mathematical properties of such graphs. We have also translated the mathematical result into efficient algorithmic ideas for recognising graphs of this class. This idea can often help in computing the inverse graph, but there is scope to improve the algorithmic idea into a concrete and precise algorithm. Further studies of properties of this class of graphs and improved algorithms for recognition are possible future directions of related research particularly for an algorithmic version of Theorem 7.6.1.

Publications

- **Journals:**

  1. M. Jadeja and R. Muthu, "Labeled Object Treemap: A New Graph-Labeling Based Technique for Visualizing Multiple Hierarchies, " in *Annals of Pure and Applied Mathematics*, vol. 13, no. 1, pp. 49–62, January 2017.

     Also appeared in *Proceedings of The 19th Japan-Korea Joint Workshop on Algorithms and Computation (WAAC 2016)*

  2. M.Jadeja and R. Muthu, "Uniform Set Labelling Vertices To Ensure Adjacency Coincides With Disjointness," in *Journal of Mathematical and Computational Science*, vol. 7, no.3, pp. 537–553, May 2017.

- **Conferences/Workshops:**

  1. M. Jadeja and K. Shah, "Tree-Map: A Visualization Tool for Large Data," in *Proceedings of 1st International Workshop on Graph Search and Beyond @SIGIR(GSB 2015)*, Santiago, Chile, 2015, pp. 9–13.

  2. R. Goyal, M. Jadeja and R. Muthu, "A New Characterisation of Total Graphs, " in *Proceedings of The 19th Japan-Korea Joint Workshop on Algorithms and Computation (WAAC 2016)*, Hakodate, Japan, 2016.

  3. M.Jadeja, R. Muthu and V. Sunitha, "Set Labelling Vertices To Ensure Adjacency Coincides With Disjointness," in *Proceedings of International Conference on Current Trends in Graph Theory and Computation (Electronic Notes in Discrete Mathematics Special Issue)*, vol. 63, pp. 237–244, December 2017.

Appendix

This is joint work together with Hitarth Kanakia.

Interactive Labeled Object Treemap (Javascript Code)

```
<html>
<!-- <script src="https://code.highcharts.com/highcharts.js"></script>
<script src="https://code.highcharts.com/modules/treemap.js"></script> -->
<script src="Queue.js"></script>
<script src="lib/jquery/dist/jquery.js"></script>
<script src="lib/bootstrap/dist/js/bootstrap.js"></script>
<link href="lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
<!-- <script
    src="lib/bootstrap-toggle-master/js/bootstrap-toggle.js"></script>
<link href="lib/bootstrap-toggle-master/css/bootstrap-toggle.css" rel =
    "styleshet" /> -->
<link rel="stylesheet" href="styles.css">
<body>
<canvas align="center" style="background-image:url('treemap.png'); "
    id="container" height="320" width="1080"></canvas>

<div style"width:800px;">
<div style="width 300px; float:left;">
<textarea id="input" rows = "5" cols="25"></textarea>
<div><button onclick = "generate()" class="btn
    btn-primary">Generate</button></div>
</div>

<div style="width 300px; float:center;">
<label><input type="checkbox" data-toggle="toggle" id="internal"
    onchange="internalChanged(this)">Internal edges </input></label>
<label><input type="checkbox" data-toggle="toggle" id="external"
    onchange="externalChanged(this)">External edges </input></label>
<label><input type="checkbox" data-toggle="toggle" id="all"
```

```
    onchange="allChanged(this)">All edges </input></label>
<label><input type="checkbox" data-toggle="toggle" id="ids"
    onchange="onlyIdsChanged(this)">IDs only</input></label>
</div>
</div>
<script>


function Node(id, type, isDummy){

this.id = id;
this.type = type;
this.isDummy = isDummy;
this.children = [];
this.IsClicked = false;
this.x = 0;
this.y = 0;
this.parent = null;
this.label = new Set();
this.color = "#FF0000";

this.setParentNode = function(node) {
this.parent = node;
}

this.getParentNode = function() {
return this.parent;
}

this.addChild = function(node) {
node.setParentNode(this);
this.children[this.children.length] = node;
}
```

```
this.getChildren = function() {
return this.children;
}

this.removeChildren = function() {
this.children = [];
}
}

var tree = [];
var group = {};

function clear(){
console.log("Clear called");
var c=document.getElementById("container");
var ctx=c.getContext("2d");
// alert('Gonna clear!');
ctx.clearRect(0,0,c.width,c.height);
}

function generate(){
var str=document.getElementById("input").value.split("\n");
clear();
console.log('Input string: '+str)
tree = [];
group = {};
tree[0] = new Node(0, str[0].split(" ")[1], false);
for(var i = 1;i<str.length;i++){
var des = str[i].split(" ");
if(des.length<2)
break;
var newNode = new Node(i,des[1],false);
```

```
tree[i] = newNode;
console.log(des);
console.log(parseInt(des[0]));
tree[parseInt(des[0])].addChild(newNode);
}
var n = tree.length;
var root = tree[0];

var obj = findMaxDegree(root);

console.log('maxDegree: '+obj.maxDegree);
console.log('levels: '+obj.levels);
fillTree(obj.maxDegree, obj.levels, n, root);
giveLabels(root,obj.maxDegree);
printLabels(root);

var q = new Queue();

q.enqueue(root);

while(!q.isEmpty()){
var cur = q.dequeue();
if(!cur.isDummy){
if(group[cur.type]==null)
group[cur.type] = [];

group[cur.type].push(cur);
}
for(var i = 0;i<cur.children.length;i++){
q.enqueue(cur.children[i]);
}
}
```

```
draw({});
}

function draw(config){
clear();
var root = tree[0];
var fromId = config.fromId;
var c=document.getElementById("container");
var ctx=c.getContext("2d");

//Color all nodes red
for(var key in tree){
tree[key].color = "#FF0000";
}

//All black
if(config.all){
for(var key in tree){
tree[key].color = "#000000";
}
}
else{
if(config.fromId != null){
tree[fromId].color = "#000000";
if(config.all){
tree[fromId].parent.color = "#000000";
}
if(config.internal){
if(tree[fromId].parent && tree[fromId].parent.type == tree[fromId].type){
tree[fromId].parent.color = "#000000";
}
}
if(config.external){
```

```
if(tree[fromId].parent && tree[fromId].parent.type != tree[fromId].type){
tree[fromId].parent.color = "#000000";
}
}

for(var i = 0;i<tree[fromId].children.length;i++){

if(config.internal && tree[fromId].children[i].type==tree[fromId].type){
console.log('black!!');
tree[fromId].children[i].color = "#000000";
}
if(config.external && tree[fromId].children[i].type!=tree[fromId].type){
tree[fromId].children[i].color = "#000000";
}
}
}
}

//Draw for B
var x = 65;
var y = 20;
for(var i = 0;group['B'] && i<group['B'].length;i++){
group['B'][i].x = x;
group['B'][i].y = y;
var next = getNextPoint('B', x,y);
x = next.x;
y = next.y;
}

//Draw for D
x = 340;
y = 30;
for(var i = 0;group['D'] && i<group['D'].length;i++){
```

```
group['D'][i].x = x;
group['D'][i].y = y;
var next = getNextPoint('D', x,y);
x = next.x;
y = next.y;
}

//Draw for E
x = 370;
y =205;
for(var i = 0; group['E'] && i<group['E'].length;i++){
group['E'][i].x = x;
group['E'][i].y = y;
var next = getNextPoint('E', x,y);
x = next.x;
y = next.y;
}

//Draw for F
x = 900;
y = 20;
for(var i = 0; group['F'] && i<group['F'].length;i++){
group['F'][i].x = x;
group['F'][i].y = y;
var next = getNextPoint('F', x,y);
x = next.x;
y = next.y;
}

for(var type in group){
for(var i = 0;i<group[type].length;i++){
ctx.fillStyle = group[type][i].color;
ctx.beginPath();
```

```
ctx.arc(group[type][i].x,group[type][i].y,5,0,2*Math.PI);
ctx.fill()
ctx.closePath();

ctx.fillStyle = "#000000";
ctx.textAlign = "center";
ctx.font = "8px Arial";

ctx.fillText(getStringFromSet(group[type][i].label),group[type][i].x,group[type][i].y+15)
}
}
ctx.fillStyle = "#000000";

if(config.all){
for(var k1 in tree){
for(var i = 0;i<tree[k1].children.length;i++){
if(!tree[k1].children[i].isDummy){
ctx.beginPath();
ctx.moveTo(tree[k1].x,tree[k1].y);
ctx.lineTo(tree[k1].children[i].x, tree[k1].children[i].y);
ctx.stroke();
ctx.closePath();
}
}
}
}
else{
for(var key in tree){
if(key!=fromId){
if(tree[key].color==="#000000"){
ctx.beginPath();
ctx.moveTo(tree[fromId].x,tree[fromId].y);
ctx.lineTo(tree[key].x,tree[key].y);
```

```
ctx.stroke()
ctx.closePath();
ctx.fillText(tree[key].id,tree[key].x-5,tree[key].y+25);
}
}
else{
ctx.fillText(tree[key].id,tree[key].x-5,tree[key].y+25);
}
}
}
}

function drawWithOnlyIds(){

clear();
var root = tree[0];

var c=document.getElementById("container");
var ctx=c.getContext("2d");

//Color all nodes red
for(var key in tree){
tree[key].color = "#FF0000";
}

//Draw for B
var x = 40;
var y = 20;
for(var i = 0;group['B'] && i<group['B'].length;i++){
group['B'][i].x = x;
group['B'][i].y = y;
var next = getNextPoint('B', x,y);
x = next.x;
```

```
y = next.y;
}

//Draw for D
x = 340;
y = 30;
for(var i = 0;group['D'] && i<group['D'].length;i++){
group['D'][i].x = x;
group['D'][i].y = y;
var next = getNextPoint('D', x,y);
x = next.x;
y = next.y;
}

//Draw for E
x = 340;
y =190;
for(var i = 0; group['E'] && i<group['E'].length;i++){
group['E'][i].x = x;
group['E'][i].y = y;
var next = getNextPoint('E', x,y);
x = next.x;
y = next.y;
}

//Draw for F
x = 900;
y = 20;
for(var i = 0; group['F'] && i<group['F'].length;i++){
group['F'][i].x = x;
group['F'][i].y = y;
var next = getNextPoint('F', x,y);
x = next.x;
```

```
y = next.y;
}

for(var type in group){
for(var i = 0;i<group[type].length;i++){
ctx.fillStyle = group[type][i].color;
ctx.beginPath();
ctx.arc(group[type][i].x,group[type][i].y,5,0,2*Math.PI);
ctx.fill()
ctx.closePath();

ctx.fillStyle = "#000000";
ctx.textAlign = "center";
ctx.font = "10px Arial";
console.log('setcheck');
console.log(group[type][i].label);
//ctx.fillText(group[type][i].id+"
    "+getStringFromSet(group[type][i].label),group[type][i].x,group[type][i].y+15);
//ctx.fillText(group[type][i].id,group[type][i].x,group[type][i].y+15);
ctx.fillText(group[type][i].id,group[type][i].x,group[type][i].y+15);
}
}
}


function findMaxDegree(root){
var maxDegree = -1;
var q = [];
q[0] = new Queue();
q[1] = new Queue();
q[0].enqueue(root);
var p = 0;
var level = 0;
```

```
while(!q[p].isEmpty()){
var next = (p+1)%2;
level++;
while(!q[p].isEmpty()){
var cur = q[p].dequeue();
maxDegree = Math.max(maxDegree, cur.children.length);
for(var i = 0;i<cur.children.length;i++){
q[next].enqueue(cur.children[i]);
}
}
p = next;
}
return {maxDegree:maxDegree, levels:level};
}

function fillTree(maxDegree, maxLevel, n, root){
var q = [];
q[0] = new Queue();
q[1] = new Queue();
q[0].enqueue(root);
var p = 0;
var nextId = n;
var count = 0;
var max = 1000;
var level = 0;
while(!q[p].isEmpty() && count++<max){
level++;
var next = (p+1)%2;

while(!q[p].isEmpty()){
var cur = q[p].dequeue();
// console.log('id: '+cur.id);
// console.log(cur.children.length);
```

```
for(var i = 0;i<cur.children.length;i++){
q[next].enqueue(cur.children[i]);
}
while(level < maxLevel && cur.children.length < maxDegree && count++<max){
var newNode = new Node(nextId, null, true)
cur.addChild(newNode);
//    console.log(cur.id+' '+cur.children.length);
q[next].enqueue(newNode);
nextId++;
}
}

p = next;
}
}



function giveLabels(root,k){
var lminus2 = [];
var lminus1 = [];
var l = [];
var lplus1 = [];
var usn = 1;
lminus2.push(root);

root.label.add(usn);
usn++;
for(var i = 0;i<root.children.length;i++){
root.children[i].label.add(k+2);
root.children[i].label.add(usn);
usn++;
console.log('check');
for (var it = root.children[i].label.values(), val= null;
```

```
   val=it.next().value; ) {
console.log(val);
}
lminus1.push(root.children[i]);
}
usn++;


for(var i = 0;i<root.children.length;i++){
for(var j = 0;j<root.children[i].children.length;j++){
var cur = root.children[i].children[j];
l.push(cur);
cur.label.add(usn);

for(var z = 1;z<=k+1;z++){
if(!cur.parent.label.has(z)){
cur.label.add(z);
}
}
usn++;
}
}


while(l.length>0){
for(var i= 0;i<l.length;i++){
for(var j = 0;j<l[i].children.length;j++){
lplus1.push(l[i].children[j]);
}
}

if(lplus1.length==0)
break;
```

```
//Step 1
for(var i = 0;i<lplus1.length;i++){
var ancestorIndex = Math.floor(i/(k*k));
var cur = lplus1[i];

for (var it = lminus1[ancestorIndex].label.values(), val= null;
    val=it.next().value; ) {
cur.label.add(val);
}
}



var pool = [];
for(var i = 0;i<k*k;i++){
pool.push(usn+i);
}
usn+=k*k;
//Step 2
for(var i = 0;i<lminus2.length;i++){
for(var j = 0;j<pool.length;j++){
lminus2[i].label.add(pool[j]);
}
}

//Step 3
for(var i = 0;i<lplus1.length;i++){
lplus1[i].label.add(pool[i%(k*k)]);
}

//Step 4
for(var i = 0;i<l.length;i++){
var cindex = i%k;
```

```
var minRejectIndex = cindex*k;
var maxRejectIndex = cindex*k + k - 1;
for(var j = 0;j<pool.length;j++){
if(!(j>=minRejectIndex && j<=maxRejectIndex))
{
l[i].label.add(pool[j]);
}
}
}

//Change the queues
lminus2 = lminus1;
lminus1 = l;
l = lplus1;
lplus1 = [];
}
}



function printLabels(root){
var q = new Queue();
q.enqueue(root);
console.log('queue: ');
for(var i= 0;i<q.getLength();i++){
var temp = q.dequeue();
console.log(temp.id);
q.enqueue(temp);
}
while(!q.isEmpty()){
var cur = q.dequeue();
console.log("id: "+cur.id);
console.log(cur.label);
// console.log(cur.children.length);
```

```
for(var i = 0;i<cur.children.length;i++){
q.enqueue(cur.children[i]);
}
}
}

var elem = document.getElementById("container")
var elemLeft=elem.offsetLeft;
var elemTop = elem.offsetTop;

elem.addEventListener('click', function(event){
var x = event.pageX
var y = event.pageY



x = Math.floor(mapX(x));
y = Math.floor(mapY(y));
console.log('x: '+x+" , y: "+y);

var internal = document.getElementById('internal');
var external = document.getElementById('external');
var all = document.getElementById('all');
var config = {all: all.checked, internal: internal.checked, external:
    external.checked, fromId: null};

var internal = document.getElementById('internal');
var external = document.getElementById('external');
var all = document.getElementById('all');

if(internal.checked || external.checked){
for(var key in tree){
var d = distance(x,y,tree[key].x,tree[key].y);
```

```
console.log('distance: '+d)
if(d<30){
config.fromId = parseInt(key);
console.log('before calling draw');
console.log(config);
draw(config);
break;
}
}
}

})

function distance(x1, y1, x2, y2){
return Math.sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}

function internalChanged(){
var internal = document.getElementById('internal');
var external = document.getElementById('external');
var all = document.getElementById('all');
var ids = document.getElementById('ids');
var config = {all: all.checked, internal: internal.checked, external:
    external.checked, fromId: null};
if(internal.checked){
all.checked = false;
config.all = false;
ids.checked = false;
}
draw(config);
}

function externalChanged(){
```

```
var internal = document.getElementById('internal');
var external = document.getElementById('external');
var all = document.getElementById('all');
var ids = document.getElementById('ids');
var config = {all: all.checked, internal: internal.checked, external:
    external.checked, fromId: null};
if(external.checked){
all.checked = false;
config.all = false;
ids.checked = false;
}
draw(config);
}

function allChanged(){
var internal = document.getElementById('internal');
var external = document.getElementById('external');
var all = document.getElementById('all');
var ids = document.getElementById('ids');
var config = {all: all.checked, internal: internal.checked, external:
    external.checked, fromId: null};
if(all.checked){
internal.checked = false;
config.internal.checked = false;

external.checked = false;
config.external = false;

ids.checked = false;
}
draw(config);
}
```

```
function onlyIdsChanged(){
var internal = document.getElementById('internal');
var external = document.getElementById('external');
var all = document.getElementById('all');
var ids = document.getElementById('ids');
var config = {all: all.checked, internal: internal.checked, external:
    external.checked, fromId: null};

if(ids.checked){
internal.checked = false;
config.internal.checked = false;

external.checked = false;
config.external = false;

all.checked = false;
config.all = false;

drawWithOnlyIds();
}
else{
draw(config);
}
}

function mapX(x){
var x1 = 49;
var y1 = 40;
var x2 = 425;
var y2 = 340;

var y = y1 + ((y2-y1)/(x2-x1))*(x-x1);
return y;
```

```
}

function mapY(x){
var x1=28;
var y1 = 20;
var x2 = 105;
var y2 = 70;

var y = y1 + ((y2-y1)/(x2-x1))*(x-x1);
return y;
}

function getStringFromSet(set){
var arr = Array.from(set);



arr.sort(function(a,b){
return a-b;
});
var str = "{";
console.log(set);
for (var i = 0;i<arr.length;i++) {
str += arr[i]+" ";
}
str+="}"
return str;
}
function getNextPoint(type, x,y){
var next = {x: 0, y:0};
if(type=='B'){
if(y+50 <= 270){
next.x = x;
next.y = y+50;
```

```
}
else{
next.x = 230;
next.y = 30;
}
}
else if(type=='D'){
if(y+40 <= 120){
next.x = x;
next.y = y+40;
}
else{
next.x = x+140;
next.y = 20;
}
}
else if(type=='E'){
if(y+40 <= 290){
next.x =x;
next.y = y+40;
}
else{
next.x = x+140;
next.y = 175;
}
}
else if(type=='F'){
if(y+50 <= 300){
next.y = y+50;
next.x = x;
}
else {
next.x = 1040;
```

```
next.y = 20;
}
}

return next;
}

</script>

</body>


</html>
```

Bibliography

[1] Noga Alon. Covering graphs by the minimum number of equivalence relations. *Combinatorica*, 6(3):201–206, 1986.

[2] Paulo Alonso Gaona García, David Martín-Moncunill, Salvador Sánchez-Alonso, and Ana Fermoso García. A usability study of taxonomy visualisation user interfaces in digital repositories. *Online Information Review*, 38(2):284–304, 2014.

[3] Armen S Asratian, Tristan MJ Denley, and Roland Häggkvist. *Bipartite graphs and their applications*, volume 131. Cambridge University Press, 1998.

[4] F. Aurenhammer, J. Hagauer, and W. Imrich. Cartesian graph factorization at logarithmic cost per edge. *Computational Complexity*, 2:331–349, 1992.

[5] VK Balakrishnan. *Schaum's Outline of Graph Theory: Including Hundreds of Solved Problems*. McGraw Hill Professional, 1997.

[6] Lyn Bartram, Axel Uhl, and Tom Calvert. Navigating complex information with the ztree. In *Graphics Interface*, pages 11–18. Citeseer, 2000.

[7] M Behzad. A characterization of total graphs. *Proceedings of the American Mathematical Society*, 26(3):383–389, 1970.

[8] Mehdi Behzad. *Graphs and their chromatic numbers*. PhD thesis, Michigan State University, 1965.

[9] Mehdi Behzad. The total chromatic number of a graph: a survey. *Combinatorial Mathematics and its Applications*, pages 1–8, 1971.

[10] Mehdi Behzad and Heydar Radjavi. The total group of a graph. *Proceedings of the American Mathematical Society*, pages 158–163, 1968.

[11] Thomas Bladh, David A Carr, and Jeremiah Scholl. Extending tree-maps to three dimensions: A comparative study. In *Asia-Pacific Conference on Computer Human Interaction*, pages 50–59. Springer, 2004.

[12] Michael Burch and Stephan Diehl. Trees in a treemap: Visualizing multiple hierarchies. In *Electronic Imaging 2006*, pages 60600P–60600P. International Society for Optics and Photonics, 2006.

[13] Seth Chaiken and Daniel J Kleitman. Matrix tree theorems. *Journal of combinatorial theory, Series A*, 24(3):377–381, 1978.

[14] Paul Erdos, Adolph W Goodman, and Lajos Pósa. The representation of a graph by set intersections. *Canad. J. Math*, 18(106-112):86, 1966.

[15] Joan Feigenbaum, John Hershberger, and Alejandro A Schäffer. A polynomial time algorithm for finding the prime factors of cartesian-product graphs. *Discrete Applied Mathematics*, 12(2):123–138, 1985.

[16] Kenneth Field. Mapping the london 2012 olympics. *The Cartographic Journal*, 49(3):281–296, 2012.

[17] Bastian Florentz and Tilo Muecke. Unification and evaluation of graph drawing algorithms for different application domains. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, pages 475–482. IEEE, 2006.

[18] Rudolf Fritsch, Rudolf Fritsch, G Fritsch, and Gerda Fritsch. *Four-Color Theorem*. Springer, 1998.

[19] George W Furnas and Jeff Zacks. Multitrees: enriching and reusing hierarchical structure. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 330–336. ACM, 1994.

[20] Joseph A Gallian. Graph labeling. *The Electronic Journal of Combinatorics*, 1000:DS6–Dec, 2015.

[21] Frédéric Gardi. The roberts characterization of proper and unit interval graphs. *Discrete Mathematics*, 307(22):2906–2908, 2007.

[22] Michael R Gary and David S Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

[23] Martin Graham and Jessie Kennedy. A survey of multiple tree visualisation. *Information Visualization*, 9(4):235–252, 2010.

[24] F. Harary. *Graph Theory (on Demand Printing Of 02787)*. Addison-Wesley series in mathematics. Avalon Publishing, 1969.

[25] Frank Harary and Geert Prins. The block-cutpoint-tree of a graph. *Publ. Math. Debrecen*, 13:103–107, 1966.

[26] Ivan Herman, Guy Melançon, and M Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on visualization and computer graphics*, 6(1):24–43, 2000.

[27] Hugh Hind, Michael Molloy, and Bruce Reed. Colouring a graph frugally. *Combinatorica*, 17(4):469–482, 1997.

[28] Hugh Hind, Michael Molloy, and Bruce Reed. Total colouring with delta+ poly (log delta) colours. 2000.

[29] Wilfried Imrich and Iztok Peterin. Recognizing cartesian products in linear time. *Discrete mathematics*, 307(3):472–483, 2007.

[30] Wilfried Imrich and Janez Žerovnik. Factoring cartesian-product graphs. *Journal of Graph Theory*, 18(6):557–567, 1994.

[31] Mahipal Jadeja and Rahul Muthu. Labeled object treemap: A new technique for visualizing multiple hierarchies. In *Proceedings of the 19th Japan-Korea Joint Workshop on Algorithms and Computation*, 2016.

[32] Mahipal Jadeja and Rahul Muthu. Labeled object treemap: A new graph-labeling based technique for visualizing multiple hierarchies. *Ann. Pure Appl. Math*, 13:49–62, 2017.

[33] Mahipal Jadeja and Rahul Muthu. Uniform set labeling vertices to ensure adjacency coincides with disjointness. *Journal of Mathematical and Computational Science*, 7(3):537–553, 2017.

[34] Mahipal Jadeja and Kesha Shah. Tree-map: A visualization tool for large data. In *GSB@ SIGIR*, pages 9–13, 2015.

[35] Sanjiv Kapoor and Hariharan Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM Journal on Computing*, 24(2):247–265, 1995.

[36] Daniel A Keim. Information visualization and visual data mining. *IEEE transactions on Visualization and Computer Graphics*, 8(1):1–8, 2002.

[37] Erica Kolatch and Beth Weinstein. Cattrees: Dynamic visualization of categorical data using treemaps. *Project report*, 2001.

[38] Nicholas Kong, Jeffrey Heer, and Maneesh Agrawala. Perceptual guidelines for creating rectangular treemaps. *IEEE transactions on visualization and computer graphics*, 16(6):990–998, 2010.

[39] J Krausz. Démonstration nouvelle dâĂŹune théorème de whitney sur les réseaux. *Mat. Fiz. Lapok*, 50(75-85):11, 1943.

[40] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[41] Bongshin Lee. *Interactive Visualizations for Trees and Graphs*. PhD thesis, 2006.

[42] Philippe G. H. Lehot. An optimal algorithm to detect a line graph and output its root graph. *J. ACM*, 21(4):569–575, October 1974.

[43] Philippe GH Lehot. An optimal algorithm to detect a line graph and output its root graph. *Journal of the ACM (JACM)*, 21(4):569–575, 1974.

[44] Saunders Mac Lane et al. A structural characterization of planar combinatorial graphs. *Duke Mathematical Journal*, 3(3):460–472, 1937.

[45] Rahul Muthu Mahipal Jadeja and Sunitha V. Set labelling vertices to ensure adjacency coincides with disjointness. *Electronic Notes in Discrete Mathematics*, 63.

[46] FWM Mank. Cristalview-the visualization of a cristal. *Master's Thesis, Mathematics and Computer Science, Technische Universiteit Eindhoven, Eindhoven, Netherlands*, page 115, 2005.

[47] Ondrej Novak. *Visualization of large graphs*. PhD thesis, MasterâĂŹs thesis, Czech Technical University in Prague, 2002.

[48] Thian-Huat Ong, Hsinchun Chen, Wai-ki Sung, and Bin Zhu. Newsmap: a knowledge map for online news. *Decision Support Systems*, 39(4):583–597, 2005.

[49] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

[50] Rahul Muthu Ravi Goyal, Mahipal Jadeja and Brijesh Patel. A new characterisation of total graphs. In *Proceedings of the 19th Japan-Korea Joint Workshop on Algorithms and Computation*, 2016.

[51] Bruce Reed. The list colouring constants. *Journal of Graph Theory*, 31(2):149–153, 1999.

[52] Fred S Roberts. *Graph theory and its applications to problems of society*. SIAM, 1978.

[53] George Robertson, Kim Cameron, Mary Czerwinski, and Daniel Robbins. Animated visualization of multiple intersecting hierarchies. *Information Visualization*, 1(1):50–65, 2002.

[54] Nicholas D Roussopoulos. A max {m, n} algorithm for determining the graph h from its line graph g. *Information Processing Letters*, 2(4):108–112, 1973.

[55] Mithileysh Sathiyanarayanan and Nikolay Burlutskiy. Design and evaluation of euler diagram and treemap for social network visualisation. In *Communication Systems and Networks (COMSNETS), 2015 7th International Conference on*, pages 1–6. IEEE, 2015.

[56] Stefan Schlechtweg, Petra Schulze-Wollgast, and Heidrun Schumann. Interactive treemapswith detail on demand to support information search in documents. In *Proceedings of the Sixth Joint Eurographics-IEEE TCVG conference on Visualization*, pages 121–128. Eurographics Association, 2004.

[57] Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM Journal on Computing*, 26(3):678–692, 1997.

[58] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, 11(1):92–99, 1992.

[59] Evren Sirin and Fusun Yaman. Visualizing dynamic hierarchies in treemaps. 2002.

[60] Vadim G Vizing. Some unsolved problems in graph theory. *Russian Mathematical Surveys*, 23(6):125–141, 1968.

[61] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[62] Hassler Whitney. Non-separable and planar graphs. In *Classic Papers in Combinatorics*, pages 25–48. Springer, 2009.