M.Sc. Engg. Thesis

An Algorithm for Solving the Maximum Leaf Spanning Tree Problem on a Bipartite Graph with Maximum Degree Three

by

Sabbir Ahmed

Master of Science in Information and Communication Technology

Institute of Information and Communication Technology (IICT)

Bangladesh University of Engineering and Technology (BUET)

Dhaka 1000

March 2012

Abstract

This thesis presents an algorithm for solving the maximum leaf spanning tree problem on a bipartite graph with maximum degree three. The maximum leaf spanning tree of a graph G is a spanning tree where the number of leaves is maximum among all possible spanning trees of G. The maximum leaf spanning tree problem is to find a maximum leaf spanning tree T of a given graph G. This problem is *NP*-Complete for regular graphs of degree three as well as for planar graphs of maximum degree four. A variation of this problem restricts to bipartite graphs and asks, for a fixed integer k, whether or not the graph contains a spanning tree with at least k leaves in one of the partite sets. This variation of the problem remains NP-Complete for planar bipartite graphs of maximum degree four. The maximum leaf spanning tree problem has applications in the area of communication networks. But this problem for bipartite graphs of maximum degree three has not been solved yet. In this thesis, an $O(n^2)$ time algorithm for solving the maximum leaf spanning tree problem on bipartite graphs of maximum degree three is presented.

Acknowledgments

I would like to express my deep and sincere gratitude to my supervisor Dr. Md. Abul Kashem Mia, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka. I owe to him for his constant supervision, encouragement, personal guidance during the progress of my thesis. His in-depth knowledge in Theoretical Computer Science especially in Graph and Algorithm Theory and his logical way of thinking have been very helpful for the successful completion of this work. I am extremely grateful to him for his cooperation.

I would like to thank the members of my thesis committee, Prof. Dr. S. M. Lutful Kabir, Prof. Dr. Md. Saiful Islam, Prof. Dr. Md. Liakot Ali and Prof. Dr. Mohammad Zahidur Rahman for their patience in understanding my work. I also warmly thank them for their valuable suggestions.

Finally, I owe my thanks to my parents for so many things which is not related to spanning tree but encourage me a lot to complete this work.

Above all, I am grateful to Almighty Allah who gave me the strength to finish this work.

Candidate's Declaration

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

_____

Sabbir Ahmed

Candidate

Contents

The thesis entitled "**An Algorithm for Solving the Maximum Leaf Spanning Tree Problem on A Bipartite Graph with Maximum Degree Three**", submitted by Sabbir Ahmed, Roll No: M10063110P, Session: October 2006, has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Science in Information and Communication Technology on March 28, 2012.

Board of Examiners

1. _____

   Dr. Md. Abul Kashem Mia — Chairman (Supervisor)
   Professor
   Department of Computer Science and Engineering
   Bangladesh University of Engineering and Technology, Dhaka-1000

2. _____

   Dr. S.M. Lutful Kabir — Member (Ex-officio)
   Professor and Director
   Institute of Information and Communication Technology
   Bangladesh University of Engineering and Technology, Dhaka-1000

3. _____

   Dr. Md. Saiful Islam — Member
   Professor
   Institute of Information and Communication Technology
   Bangladesh University of Engineering and Technology, Dhaka-1000

4. _____

   Dr. Md. Liakot Ali — Member
   Professor
   Institute of Information and Communication Technology
   Bangladesh University of Engineering and Technology, Dhaka-1000

5. _____

   Dr. Mohammad Zahidur Rahman — Member (External)
   Professor
   Department of Computer Science and Engineering
   Jahangirnagar University, Savar, Dhaka

CHAPTER 1
Introduction

In this chapter, we provide the necessary background, present state and motivation for this study on the maximum leaf spanning tree; define the problem and scope of this thesis. We start, in Section 1.1, by giving the historical background on spanning trees. We also define maximum leaf spanning tree, connected graph, bipartite graph. We define the maximum leaf spanning tree problem for bipartite graph also in Section 1.1. Section 1.2 presents some applications of the maximum leaf spanning tree problem. Section 1.3 defines the scope and motivation of the thesis and in Section 1.4 we define the objectives of the thesis. Finally, in Section 1.5 we discuss the results obtained for solving the problems of the thesis.

1.1 Background

Graph theory is a delightful playground for the exploration of proof techniques in discrete mathematics and its results have applications in many areas of computing, social and natural sciences. Recent research effort is concentrating on evolving efficient algorithms in combinatorial mathematics especially graph problems. A graph $G = (V, E)$ with n vertices and m edges consist of a vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and an edge set $E = \{e_1, e_2, \ldots, e_m\}$, where, an edge in E joins two vertices in V. Figure 1.1 depicts a graph G of five vertices and six edges, where vertices are drawn by circles, edges by lines, vertex names next to the circles and edge names next to the lines. Efficient algorithms have been obtained for various graph problems, such as coloring problems, planarity testing problem and maximum flow problem [1].

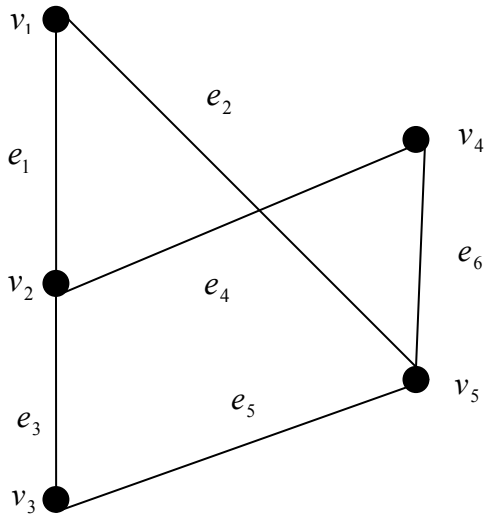


Figure 1.1 A graph *G* with five vertices and six edges.

1.1.1 The Maximum Leaf Spanning Tree (MLST) Problem

The Maximum Leaf Spanning Tree (MLST) problem is said that given a graph and a positive integer *k*, we have to say that whether there is a spanning tree consisting of at least *k* number of vertices with degree 1 [2].

1.1.2 Variant of Maximum Leaf Spanning Tree Problem for Bipartite Graph

Variant of Maximum Leaf Spanning Tree problem for bipartite graph is one of the restriction versions of the Maximum Leaf Spanning Tree problem. It is said that given a bipartite graph and a positive integer *k*, find the spanning tree with at least *k* leaves at one partite set, where the value of *k* is less than the size of the one partite set [2].

1.1.3 Set Version of Maximum Leaf Spanning Tree Problem

All the problems of Maximum Leaf Spanning Tree problem can also be represented as a new notion of "set version" [1]. Consider the input instance of Maximum Leaf Spanning Tree problem where we have an integer $k \leq V$. But in the set version the input instance *k*

is replaced by $X \subseteq \prod_T$ .Where X is a set of vertices. We have to find out the desired spanning tree. Where $\prod_T = \{v : v \text{ is a leaf node of } T\}$.

1.1.4 Set Version of Maximum Leaf Spanning Tree Problem for Bipartite graph

Maximum Leaf Spanning Tree problem for bipartite graph is also represented by the set version. The problem states that given a bipartite graph with two partite sets X, Y and there is a set $X_1 \subseteq X$. We have to find out a spanning tree such that $X_1 \subseteq \prod_T$. Where $\prod_T = \{v : v \text{ is a leaf node of } T\}$ [1].

1.2 Application of Spanning Trees

There are many interesting structures of graph that has been exploited for various problems. Paths, cycles and trees are the most basic structures of graphs. In the field of computer science these basic structures along with their various findings have contributed to a great extent to evolve many new research areas [3].

„Spanning tree" is a „spanning" version of tree that is very much interesting for extensive research. Research on this particular area is interesting because numerous practical applications involve this, some of which has been depicted next.

Electronic Circuitry Design - The popular and classical example of a spanning tree application contributes to the design of electronic circuitry. It is often necessary to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, we can use an arrangement of n–1 wires, each connecting two pins. Of all such arrangements, the one uses the least amount of wire is usually the most desirable. This is in fact the famous minimum-weight spanning tree problem [4].

Minimum Connector Problem - Suppose we wish to build a railway network connecting n given cities so that a passenger can travel from any city to another with the constraint that the total amount of track must be a minimum (in fact this should be the aim of the government building the network since minimum track would ensure minimum cost of construction). So in effect the problem is to find an efficient algorithm for deciding which of the n^{n-2} possible spanning trees uses the least amount of track [5]. There are many practical applications of spanning trees. Spanning tree comes inseparably with networking.

Indirect uses of minimum spanning tree include optimal broadcast in unreliable medium, optimal data storage in two-dimensional arrays, min-max path problems, and cluster analysis.

1.3 Scope and Motivation of the Thesis

This thesis deals with the problem of finding maximum leaf spanning tree of bipartite graphs with maximum degree three. We will present an algorithm to find out the maximum leaf spanning tree for bipartite graph of maximum degree three. This algorithm is based on the existing Breadth First Search algorithm to check the connectivity of a graph.

The maximum leaf spanning tree problem is NP-complete for regular graphs of degree three [2] as well as for planar graphs of maximum degree four [1]. The optimization version of this problem is known to be NP-hard [1]. This problem has applications in the area of communication networks [6]. For example, a connected graph $G = (V, E)$ may model a computer network, where vertices V correspond to the computers and edges E correspond to the computers that may communicate directly with one another. A spanning tree T of G provides a subset of the communication links by which any two computers may communicate, using only this subset of links. The nodes of degree one in the tree correspond to computers that performs less communication processing than nodes of degree at least two. Therefore, it is reasonable to ask for a spanning tree of G with the maximum number of leaf nodes. But this problem for bipartite graphs of maximum degree three has not been solved yet. So we solved this problem for bipartite graphs with maximum degree three.

1.4 Objectives of the Thesis

This research will focus on the following objectives:

a) Outline an optimization procedure to solve the maximum leaf spanning tree problem for bipartite graphs with maximum degree three.
b) Give an algorithm to solve the problem.
c) Verify the correctness of the algorithm.
d) Analyze the time-complexity of the proposed algorithm.

1.5 Organization of the Thesis

The organization of this thesis is as follows. Chapter 2 gives preliminary definitions and representation of maximum leaf spanning tree. Chapter 3 describes some interesting problems of spanning trees. Chapter 4 gives an algorithm for solving the maximum leaf spanning tree problem for bipartite graph of maximum degree three and determines its time-complexity. Chapter 5 concludes with a discussion of the result and future works. Following Table 1.1 shows the known results of MLST problem.

Table 1.1 Algorithms for MLST problem of bipartite graph.

Graphs	**Time**	**References**
Bipartite graph with maximum degree four	*NP*-Complete	Ben.Li *et al.* Information processing letter, 2006 [2].
Cubic graph	*NP*-Complete	P. Lemke *et.al* IMA publication No428, 1988 [6].
Regular graph with degree four	*NP*-Complete	A.J Storer *et.al* Information processing letter, 1981 [5].
Bipartite graph with maximum degree three	$O(n^2)$	Ours

CHAPTER 2

Preliminaries

In this chapter, we define some basic definitions and some special types of graphs. Definitions that are not given here are discussed as they are needed. In Section 2.1, we start by giving the definitions of some basic terms of graphs which are related to and used throughout the thesis. Section 2.2 defines some necessary terms on algorithms and complexity. We give a precise introduction to breadth-first search, depth-first search, matching and set covering in Section 2.3. Section 2.4 defines maximum leaf spanning tree and bipartite graph.

2.1 Basic Terminology

In this section we give some definitions of standard graph-theoretical terms used throughout this thesis.

2.1.1 Graphs and Multigraphs

A graph G is a triple consisting of a vertex set $V(G)$, an edge set $E(G)$, and a relation that associates each edge with two vertices (not necessarily distinct) called endpoints. It is a structure (V, E), which consists of a finite set of vertices V and a finite set of edges E; each edge is an ordered pair of distinct vertices. We denote the set of vertices of G by $V(G)$ and the set of edges by $E(G)$. Figure 2.1 depicts a graph G, where each vertex in $V(G) = \{v_1, v_2, ..., v_7\}$ is drawn by a small black circle and each edge in $E(G) = \{e_1, e_2, ..., e_7\}$ is drawn by a line segment. Throughout this thesis the number of vertices of G is denoted by n, that is, $n = | V |$, and the number of edges of G is denoted by m, that is, $m = |E|$. Thus for the graph in Figure 2.1 $n = 7$ and $m = 7$. In a graph *multiple edges* join the same pair of vertices, while a *loop* joins a vertex to itself. The graph in which loops and multiple edges are allowed is called a multigraph [8]. A graph G having no multiple edges and loops, is called a simple graph. Sometimes they are referred to simply as graphs. In our thesis we consider graphs free from

multiple edges and loops. We denote an edge between two vertices u and v of G by (u, v). If $(u, v) \in E$, then two vertices u and v of graph G are said to be adjacent; edge (u, v) is then said to be incident to vertices u and v; u is neighbor of v.The degree of a vertex v in G is the number of edges incident to v, and is denoted by $d_G(v)$ or simply $d(v)$. Vertices v_1 and v_2 in Figure 2.1 are adjacent, and $d(v_1) = 2$, since two edges e_1 and e_2 are incident to v_1.

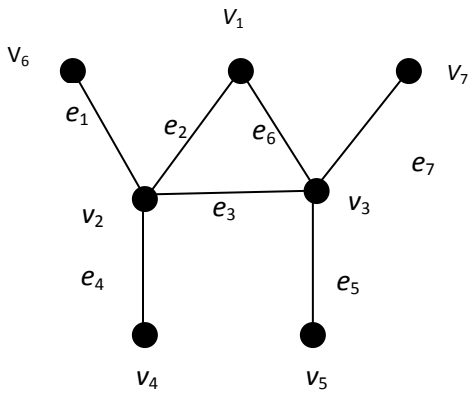


Figure 2.1 A Graph with seven vertices and seven edges.

2.1.2 Subgraphs

A subgraph of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$, then $G' \subseteq G$. If G' contains all the edges of G that join two vertices in V', then G' is said to be the subgraph induced by V'. Figure 2.2 depicts a subgraph of G in Figure 2.1 induced by $\{v_1, v_2, v_3, v_4, v_5\}$. A spanning subgraph of a graph $G = (V, E)$ is a subgraph of G such that $V' = V$ and $E' \subseteq E$.

We often construct new graphs from old ones by deleting some vertices or edges. If v is a vertex of a given graph $G = (V, E)$, then $G - v$ is the subgraph of G obtained by deleting the vertex v and all its incident edges. More generally, if V' is a subset of V, then $G - V'$ is the subgraph of G obtained by deleting by V' and all the edges incident to V'. Then $G - V'$ is a subgraph of G induced by $V - V'$[9].

Similarly, if e is an edge of G, then $G - e$ is the subgraph of G obtained by deleting the edge e. More generally, if $E' \subseteq E$, then $G - E'$ is the subgraph of G obtained by deleting the edges in E'.

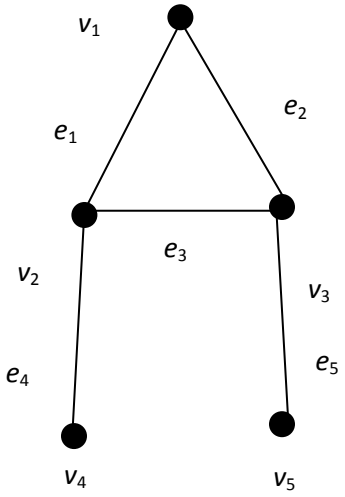
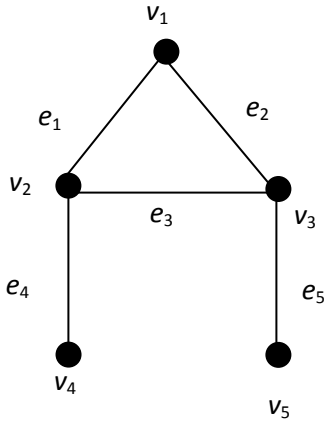


Figure 2.2 A vertex-induced subgraph.

2.1.3 Connectivity

A graph G is *connected* if for every pair $\{u, v\}$ of distinct vertices, there is a path between u and v. A graph, which is not connected, is called a *disconnected* graph. A (connected) component of a graph is a maximal connected subgraph.



(a)

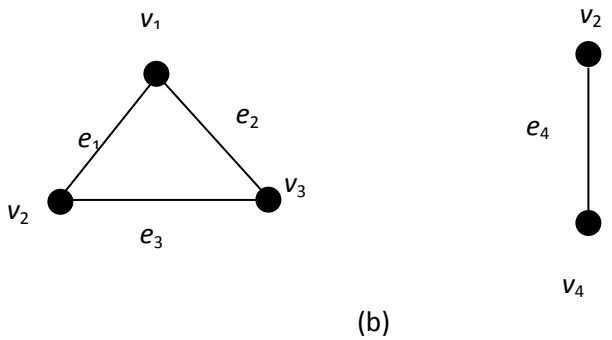


Figure 2.3 (a) A connected graph, (b) a disconnected graph with two connected component.

The graph in Figure 2.3(a) is a connected graph since there is a path for every pair of distinct vertices of the graph. Figure 2.3(b) shows a disconnected graph where there is no path from vertex v_1 to vertex v_2 and hence the graph is disconnected. The graph has two disjoint connected components.

The connectivity $\kappa(G)$ of a graph G is the minimum number of vertices whose removal results in a disconnected graph or a single-vertex graph K_1. We say G is k-connected if $k \leq \kappa(G)$. We call a set of vertices in a connected graph G a *separator* or a *vertex-cut* if the removal of the vertices in the set results in a disconnected or single-vertex graph. If a vertex-cut contains exactly one vertex then we call the vertex a *cut vertex*.

2.1.4 Paths and Cycles

A v_0-v_l *walk*, v_0, e_1, v_1, …, v_{l-1}, e_l, v_l in G is an alternating sequence of vertices and edges of G, beginning and ending with a vertex, in which each edge is incident to two vertices immediately preceding and succeeding it. If the vertices v_0, v_1, …, v_l are distinct (expect possibly v_0, v_l), then the walk is called a *path* and usually denoted either by the sequence of vertices v_0, v_1, …, v_l or by the sequence of edges e_0, e_1, …, e_l. The length of the path is l, one less than the number of vertices. A path or walk is closed if $v_0 = v_l$. A closed path containing at least one edge is called a *cycle*.

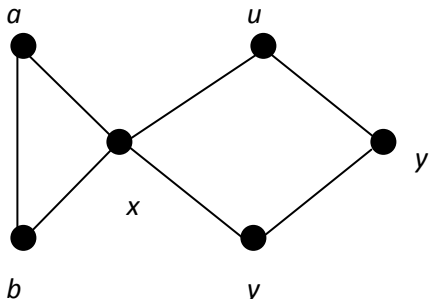


Figure 2.4 Walk, trail, cycle, and path.

In Figure 2.4, (a, x, a, x, u, y, v, x, b, a) is a closed odd walk of length 9. Now omitting first two steps yields a closed trail (no edge repetition). The edge set of this trail is the union of the edge sets of 2 pair wise edge-disjoint cycle: (a, x, b, a) and (x, u, y, v, x). Finally, the u, v-trail (u, x, a, b, x, v) contains the edges of the u, v-path (u, x, v).

2.1.5 Trees

A *tree* is a connected acyclic graph. Figure 2.5 is an example of a tree.

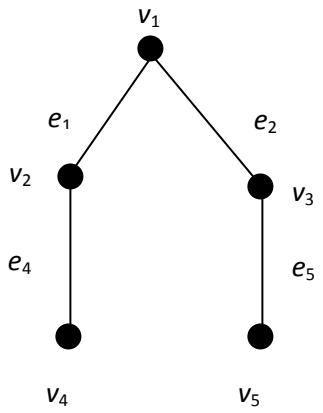


Figure 2.5 A tree.

A rooted tree is a tree in which one of the nodes is distinguished from the others. The distinguished node is called the *root* of the tree. The root of the tree is generally drawn at the top. In Figure 2.5, the root is v_1. Every node u other than the root is connected by an edge to some other node v called the parent of u. We also call u a child of v. In Figure 2.5 v_1 is the parent of v_2 and v_3 while v_2 is the parent of v_4 and v_3 is the parent of v_5. v_4 and v_5, are the children of v_2 and v_3, respectively. Root v_1 has two children, which are v_2 and v_3. A leaf is a node of a tree that has no child. An internal node is a node that has one or more children. Thus every node of a tree is either an internal node or a leaf. The degree of a leaf is always one, that is for a leaf $d(v) = 1$. In Figure 2.5, v_4 and v_5 are the leaves.

A *spanning tree* of $G = (V, E)$ is a spanning subgraph of G which is a tree. Figure 2.5 depicts a spanning tree of the graph in Figure 2.2.

2.1.6 Forest and Spanning Tree

A forest is an acyclic graph that is each of the connected components of a forest is a tree. Figure 2.6 shows a forest with three components (trees).

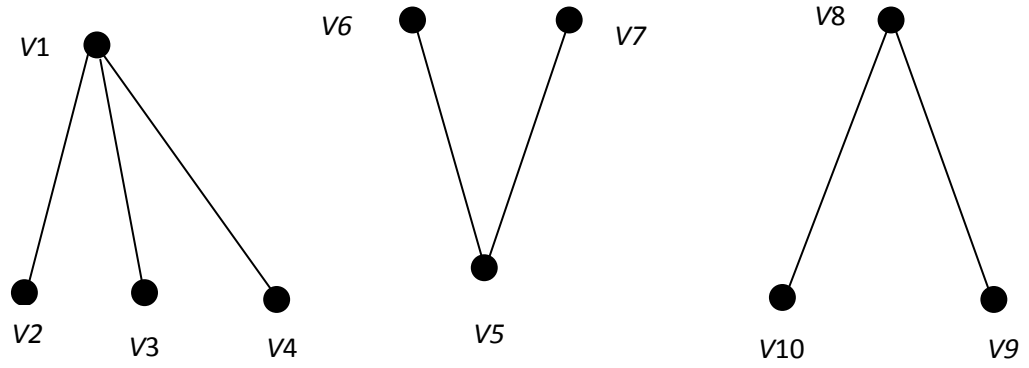


Figure 2.6 A forest with three components.

A spanning tree is a spanning subgraph that is a tree. A spanning subgraph of a graph G is a subgraph with vertex set $V(G)$. Figure 2.7 shows a spanning tree of Figure 2.1. Spanning trees

have been found to be structures of paramount importance also in practical applications such as in network and communication problems as so on [10].

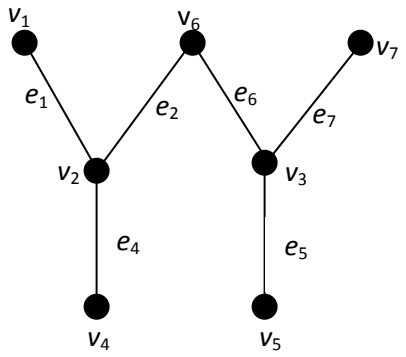


Figure 2.7 A spanning tree of graph *G* in Fig. 2.1.

2.1.7 Bipartite Graph

A bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets *U* and *V* such that every edge connects a vertex in *U* to one in *V*; that is, *U* and *V* are independent sets. Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.

The two sets *U* and *V* may be thought of as a coloring of the graph with two colors: if we color all nodes in *U* blue, and all nodes in *V* green, each edge has endpoints of differing colors, as is required in the graph coloring problem. In contrast, such a coloring is impossible in the case of a nonbipartite graph, such as a triangle: after one node is colored blue and another green, the third vertex of the triangle is connected to vertices of both colors, preventing it from being assigned either color. One often writes $G = (U, V, E)$ to denote a bipartite graph whose partition has the parts *U* and *V*. If $|U| = |V|$, that is, if the two subsets have equal cardinality, then *G* is called a *balanced* bipartite graph [1, 2, 6, 11]. Figure 2.8 shows a bipartite graph.

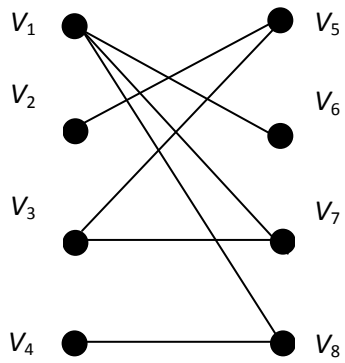


Figure 2.8 A bipartite graph.

A complete bipartite graph, $G = (V_1 + V_2, E)$, is a bipartite graph such that for any two vertices, $v_1 \in V_1$ and $v_2 \in V_2$, v_1v_2 is an edge in G. The complete bipartite graph with partitions of size $|V_1|=m$ and $|V_2|=n$, is denoted $K_{m,n}$.

2.1.8 Degree of a Vertex

The degree of a vertex v in a graph G is the number of edges incident to v, and is denoted by $d(v)$. The maximum degree of G is denoted by $\Delta(G)$ or simply by Δ. In Figure 2.1, the degree of vertex v_1 is 2 and the maximum degree Δ of G, is 4 as $d(v_2)$ and $d(v_3)$ is 4. A vertex of degree 0 is called an *isolated* vertex.

2.1.9 Complete Graph and Cliques

A *complete graph* is a simple graph whose vertices are pair wise adjacent. A complete graph with n vertices is denoted as k_n. A *clique* in a graph is a set of pair wise adjacent vertices. A complete graph has many subgraphs that are not cliques, but every induced subgraph of a complete graph is a clique. Figure 2.9(a) is both a complete graph k_6 and also a clique of size 6. Subgraph with $\{v_1, v_2, v_4, v_5\}$ in Figure 2.9 (b) is a clique of size 4.

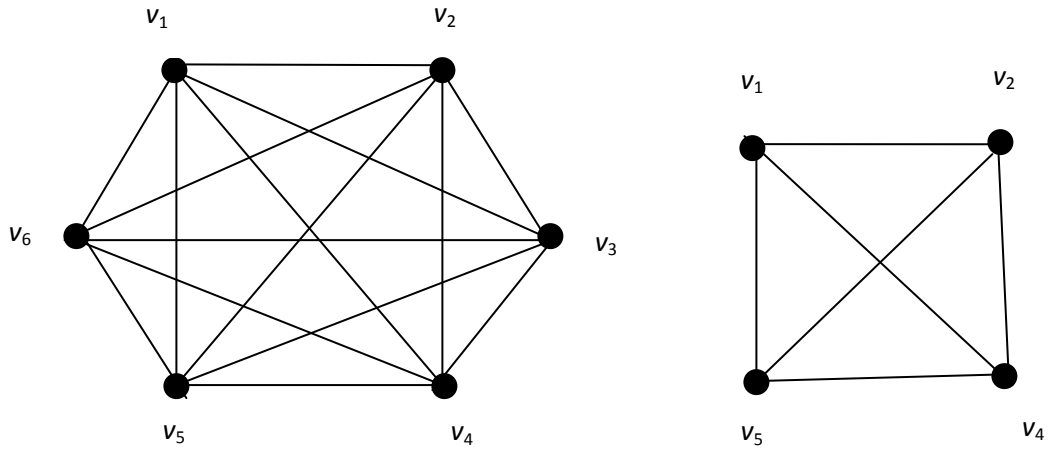


Figure 2.9 (a) A complete graph k_6 and (b) subgraph with $\{v_1, v_2, v_4, v_5\}$ is a clique.

2.1.10 Connected Components and Separators

A graph G is connected if for every pair $\{u, v\}$ of distinct vertices there is a path between u and v. A (connected) component of a graph is a maximal connected subgraph [12]. A graph which is not connected is called a *disconnected* graph. Figure 2.10 shows a disconnected graph with three (connected) components.

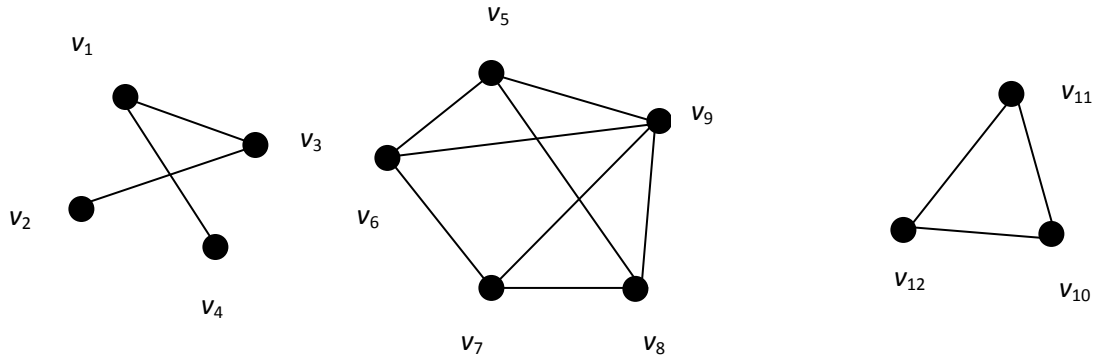


Figure 2.10 A graph G with three components.

Separation of a graph can be done in two ways: using vertex separator or using edge separator. A separator disconnects a graph into more than one components. A vertex separator of a connected graph G is a set of vertices whose deletion disconnects G. The graph G in

Figure 2.11(a) has a separator $\{v_3, v_4\}$. An edge separator of a connected graph G is a set of edges whose deletion disconnects G.

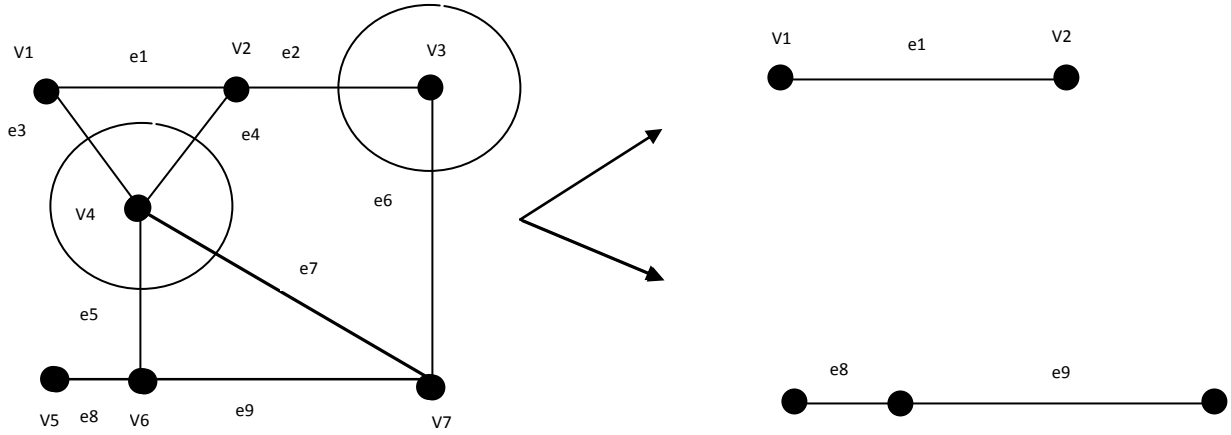


Figure 2.11 Separation of a graph G with a vertex separator.

2.2 Algorithms and Complexity

In this section we briefly introduce some terminologies related to complexity of algorithms. The most widely accepted complexity measure for an algorithm is the running time, which is expressed in terms of operations it performs for producing the final output.

2.2.1 The Notation $O(n)$

In analyzing the complexity of an algorithm, we often interested only in the asymptotic behavior, that is, the behavior of the algorithm when applied to very large inputs. To deal with such a property of functions, we shall use the following notations for asymptotic notations for asymptotic running time. Let $f(n)$ and $g(n)$ be the functions from the positive integers to the positive reals, then we write $f(n) = O(g(n))$ if there exists positive constants c_1 and c_2 such that $f(n) \leq c_1 g(n) + c_2$ for all n [4,13].

2.2.2 Polynomial Algorithms

An algorithm is said to be polynomial bounded (or simply polynomial) if its complexity is bounded by a polynomial of the size of a problem instance. Examples of such complexities are $O(n)$, $O(n\log n)$, $O(n^{100})$ etc. The remaining algorithms are usually referred to as exponential or non-polynomial. Examples of such complexity are $O(2^n)$, $O(n!)$ etc. When the running time of an algorithm is bounded by $O(n)$, we call it a linear-time algorithm or simply a linear algorithm [14].

2.2.3 Complexity Classes: *P* and *NP*

A problem is said to have a polynomial-time algorithms if the worst case running time is $O(n^k)$ for input size n and for some constant k. Generally, problems that are solvable by polynomial-time algorithms are tractable or easy, and problems that require super polynomial time are intractable or hard. Based upon the running time of algorithms, next we define complexity classes. The class *P* consists of all those decision problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input; while the class *NP* consists of all those problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a non-deterministic machine. Any problem in *P* is also in *NP*, since if a problem is in *P* then we can verify it in polynomial time [13].

2.2.4 *NP*-Complete Problem

Here, we are mainly interested in another class of problems, called *NP-Complete (*or *NPC) problems*, which can be loosely described as the hardest problems in *NP* and therefore they are the least likely to be in *P* [2, 6]. No polynomial-time algorithm has yet been discovered for an *NP*-Complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any of them.

More precisely, a decision problem *C* is *NP*-Complete if it is complete for *NP*, meaning that:

(1) it is in *NP*, and

(2) it is *NP*-hard, i.e. every other problem in *NP* is polynomial-time reducible to it.

Here "polynomial-time reducible" means that for every problem L, there is a polynomial-time many-one reduction, a deterministic algorithm which transforms instances $l \in L$ into instances $c \in C$, such that the answer to c is YES if and only if the answer to l is YES. To prove that an *NP* problem A is in fact an *NP*-Complete problem it is sufficient to show that an already known *NP*-Complete problem reduces to A. A consequence of this definition is that if we had a polynomial-time algorithm for C, we could solve all problems in *NP* in polynomial time.

2.2.5 Approximation Algorithm

Many problems of practical significance are *NP*-Complete but are too important to abandon merely because obtaining an optimal solution is intractable. If a problem is *NP*-Complete, we are unlikely to find a polynomial time algorithm for solving it exactly, but even so, there is still hope. There are at least three approaches to getting around *NP*-Completeness.

- First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
- Second, we may be able to isolate important special cases that are polynomial time solvable.
- Third, the chances of finding a near-optimal solution in polynomial time still remain; it may be in either worst case or average case.

In practice near optimality is often desirable where exact solution is far beyond. An algorithm that returns near optimal solutions is called an approximation-algorithm.

At present, all known algorithms for *NP*-Complete problems require time that is super polynomial in the input size [14]. It is unknown whether there are any faster algorithms. Therefore, to solve an *NP*- Complete problem for any nontrivial problem size, generally it may still be possible to find near - optimal solutions in polynomial time. An algorithm that quickly finds a suboptimal solution that is within a certain (known) range of the optimal one is called an approximation algorithm.

2.3 Searching, Matching and Set Cover

In this section we give the basic idea of search techniques like breadth-first search, depth-first search. They are very important to ensure the connectivity of a graph. We also present notion about matching and set covering.

2.3.1 Depth-First Search

The strategy followed by depth-first search (DFS) is, as its name implies, to search "deeper" in the graph whenever possible. In DFS edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered, this process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is continued from that source. This entire process is repeated until no vertex is left discovered.

The running time of DFS is $O(n+m)$, where n is the number of vertices and m is the number of edges of the graph G.

2.3.2 Breadth-First Search

Breadth-first search is one of the simplest algorithms for searching a graph and the *archetype* for many important graph algorithms. Prim's minimum spanning tree algorithm and Dijkstra's single source shortest path algorithm use similar idea to those in breadth first search.

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth first search systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also computes a breadth-first tree with root s that contains all reachable vertices [2]. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

The running time of the breadth-first algorithm is $O(n+m)$, where n is the number of vertices and m is the number of edges, in G.

2.3.3 Matching

A matching in a graph G is a set of non-leaf edges with no shared endpoints. The vertices incident to the edges of a matching M are saturated by M; the others are unsaturated. A perfect matching in a graph is a matching that saturates every vertex. Figure 2.12 shows a matching. Let us consider $K_{n,n}$ with partite sets $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$. A perfect matching defines a bijection from X to Y [15].

Each matching is represented by a permutation of $[n]$, mapping i to j, when x_i is matched to y_j. We can express the matchings as matrices. With X and Y indexing the rows and columns, we let position i, j be 1 for each edge $x_i\, y_j$ in a matching M to obtain the corresponding matrix. There is one 1 in each row and each column.

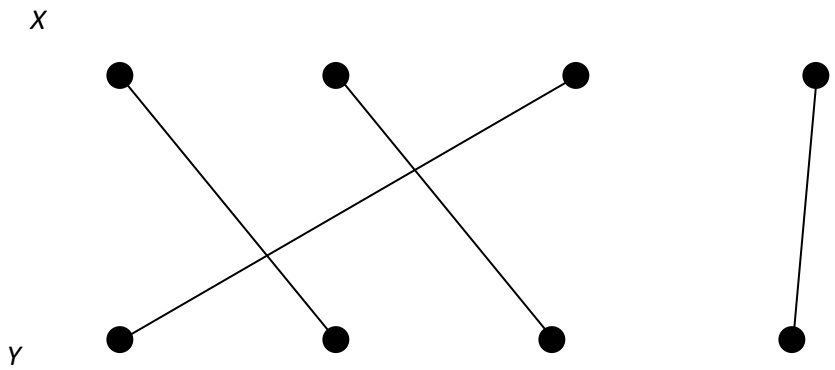


Figure 2.12 Matching.

2.3.3.1 Maximum Matching

A matching is a set of edges, so its sizes is the number of edges. We can seek a large matching by iteratively selecting edges whose endpoints are not used by the edges already selected, until no more are available. This yields a maximal matching but may not be a maximum matching. A maximal matching in a graph is a matching that cannot be enlarged by adding an edge. A maximum matching is a matching of maximum size among all matchings in the graph. A matching M is maximal if every edge not in M is incident to an edge already in M. Every maximum matching is a maximal matching, but the converse need not hold.

2.3.3.2 Maximum Bipartite Matching

To find a maximum matching, we iteratively seek augmenting paths to enlarge the current matching. In a bipartite graph, if we do not find an augmenting, we will find a vertex cover with the same size as the current matching has maximum size.

2.3.4 Set Cover

The set-cover problem is an optimization problem that models many resource-selection problems.

An instance (X, F) of the set-cover problem consists of a finite set X and a family of subsets F of X, such that every element of X belongs to at least one subset in F. In Figure 2.13, Consider an instance (X, F) of the set cover problem, where X consists of the 12 black points *and* $F = \{S_1, S_2, S_3, S_4, S_5, S_6\}$ A minimum size set cover is $C = \{S_3, S_4, S_5\}$ [1].

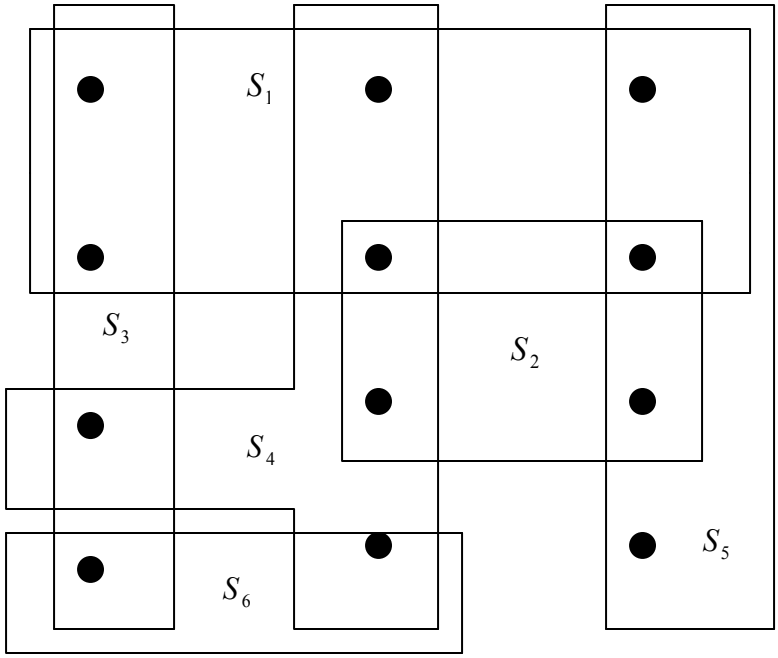


Figure 2.13 An instance (*X, F*) of the set cover problem.

CHAPTER 3

Some Interesting Problems on Spanning Trees

Spanning trees of a connected graph have always been the focus of extensive research. Spanning trees with various constraints and restricted conditions seem to pose various interesting problems. Here we describe some spanning tree problems by imposing various constraints and restrictions on graph parameters. Spanning trees with restrictions on the number of leaves have applications in communication networks and circuits layouts.

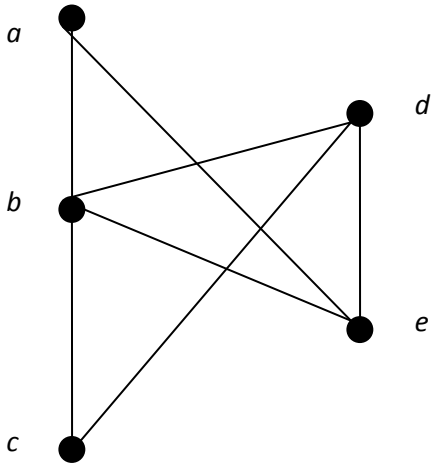
3.1 Decision Problems on Spanning Trees

In this section we describe some problems and theorems on spanning trees. First we describe problems on general graphs and next we describe some problems and theorems on bipartite graphs.

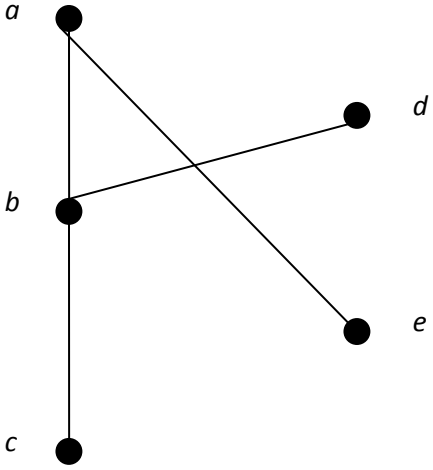
3.1.1 Degree Constrained Spanning Tree Problem

Definition: Given a connected graph $G = (V, E)$ and a positive integer $K < |V|$, we are asked the question whether there is a spanning tree T_G of G such that no vertex in T has degree larger than k [2].

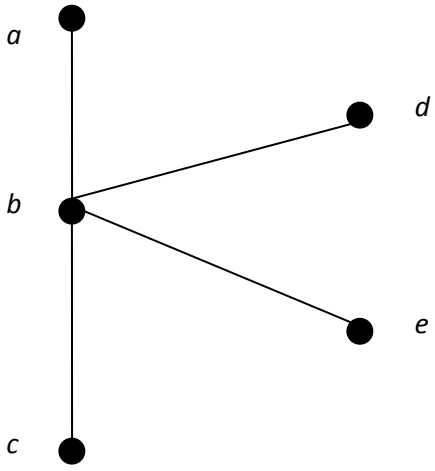
Example: Consider a graph G in Figure 3.1. It has five nodes namely $V = \{a, b, c, d, e\}$ and given edge in $E = \{(a, b), (a, e), (b, c), (b, d), (b, e), (c, d), (d, e)\}$. Also assume that the value of k is given to be 3. Now in Figure 3.1(b) and (c) there are two spanning trees T_1 and T_2. T_2 has a vertex which has degree larger than k. So, it is not our desired spanning tree. In T_2, there is no vertex which has degree larger than 3. So it is our desired spanning tree.



(a) An arbitrary graph *G*.



(b) A spanning tree T_1 of *G*.



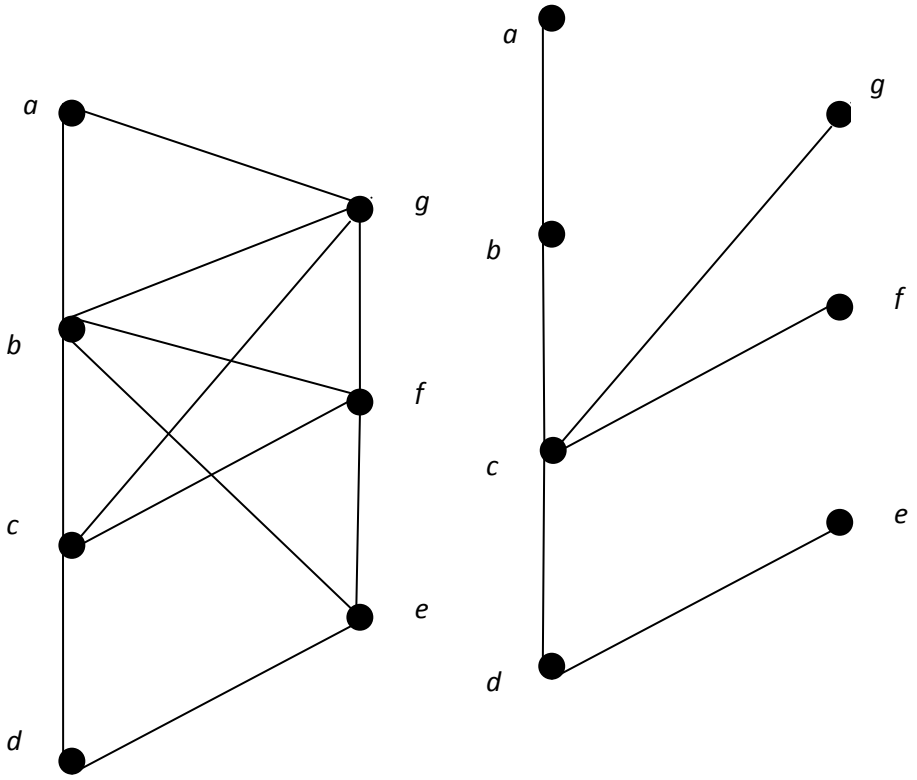
(c) A spanning tree T_2 of *G*.

Figure 3.1 Degree Constrained Spanning Tree.

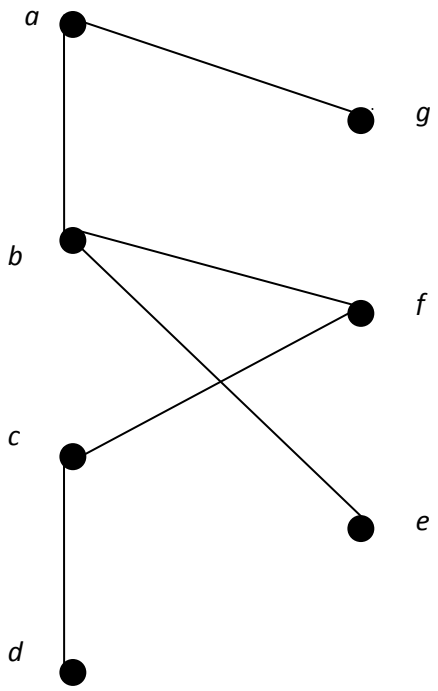
3.1.2: Minimum Leaf Spanning Tree Problem

Definition: Given a connected graph $G = (V, E)$ and a positive integer $K < |V|$, we are asked the question whether there is a spanning tree T_G such that K or less vertices have degree 1 [1, 2].

Example: Consider a graph $G = (V, E)$ in Figure 3.2, where $V = \{a, b, c, d, e, f, g\}$ and $E = \{(a, b), (a, g), (b, c), (b, e), (b, f), (c, d), (c, f), (c, g), (d, e)\}$. We also assume that the value of k in the input instance is 3. Now we find two spanning trees namely T_1 and T_2 of G. In T_1, the edges are $\{(a, b), (b, c), (c, d), (c, f), (c, g), (d, e)\}$ and the number of leaves or the vertices with degree 1 is 4. So it is not our desired spanning tree. On the other hand in T_2, the edges are $\{(a, b), (a, g), (b, e), (b, f), (c, d), (c, f)\}$. It has 3 vertices which has degree 1. So it is our desired spanning tree.



(a) An arbitrary graph G. (b) A spanning tree T_1 of G.



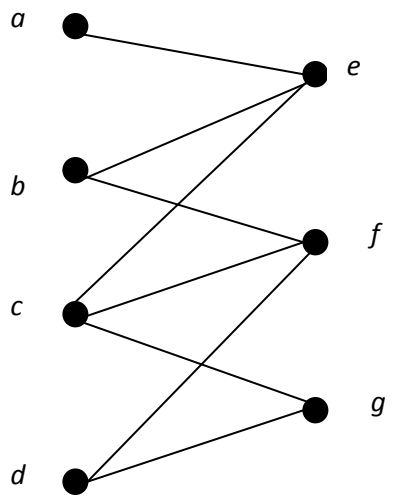
(c) A spanning tree T_2 of G.

Figure 3. 2 Minimum leaf spanning tree.

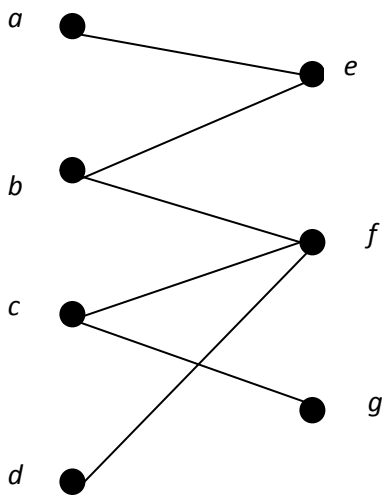
3.1.3 Maximum Leaf Spanning Tree Problem

Definition: Let $G = (V, E)$ be a connected graph and $K < |V|$ be a positive integer. We are asked whether G contains a spanning tree T_G consisting of at least k vertices of degree 1[1, 2].

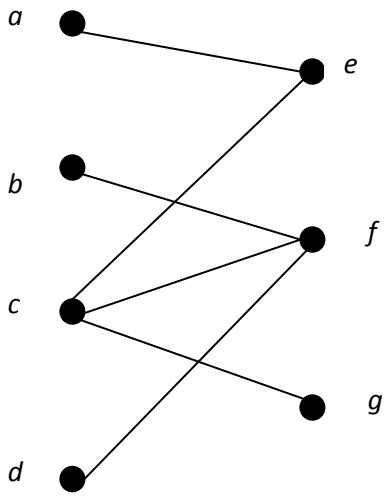
Example: Consider a graph G in Figure 3.3. Its vertices are $V = \{a, b, c, d, e, f, g\}$ and $E = \{(a, e), (b, e), (b, \text{f}), (c, e), (c, f), (c, g), (d, f), (d, g)\}$. Also suppose that the value of K in the input instance is 4. From the graph we found two spanning trees namely T_1 and T_2. In T_1 there are 3 vertices which have degree 1. So it is not our desired tree. But in T_2 there are 4 vertices which have degree 1. So it is our desired tree.



(a) An arbitrary graph *G*.



(b) A spanning tree T_1.



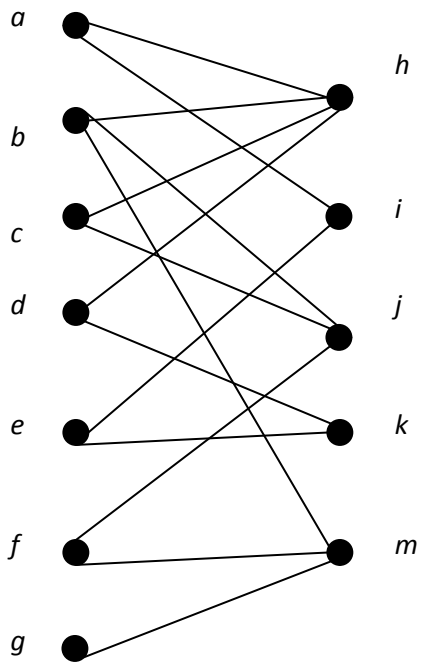
(c) A spanning tree T_2 of *G*.

Figure 3. 3 Maximum leaf spanning tree.

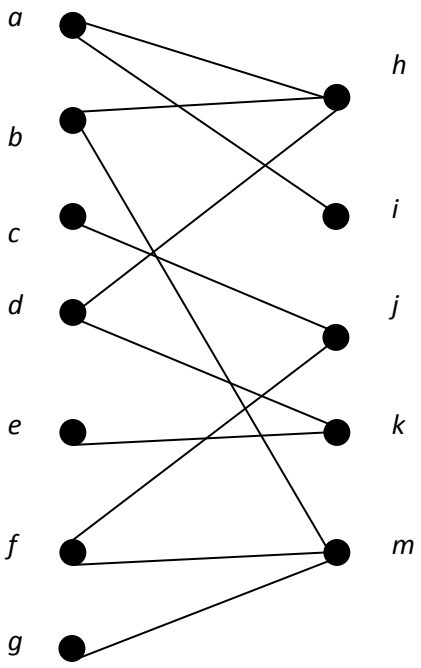
3.1.4 Variant of Maximum Leaf Spanning Tree Problem for Bipartite Graph

Definition: Given a connected graph $G = (V, E)$ with partite sets X and Y and a positive integer $K < |X|$, we are asked the question of whether there is a spanning tree T_G of G such that the number of leaves in T_G belonging to X is greater than or equal to K [4].

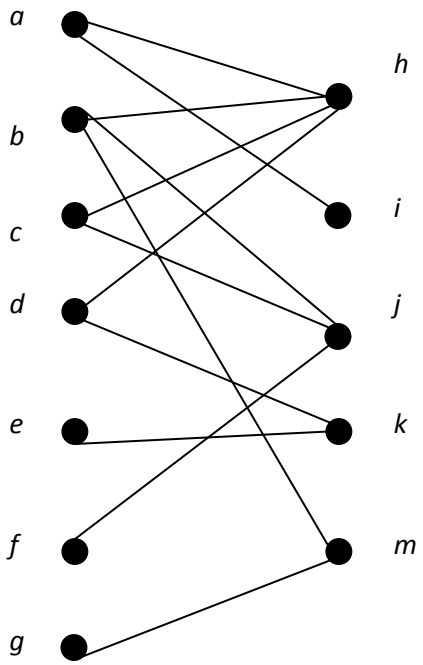
Example: Consider the bipartite graph G in Figure 3.4 (a) with partite sets $X = \{a, b, c, d, e, f, g\}$ and Y. Also assume that the value of K in the input instance is given to be 3. Now in Figure 3.4(b) and (c), there are two spanning trees T_1 and T_2 of G. Since in T_x, the number of leaves belonging to X is only 2 (namely *e, c* and *g*), it is not a desired spanning tree. On the other hand T_2 is a desired spanning tree of G for the given value of K(=3).



(a) An arbitrary graph G.



(b) A spanning tree T_1 of G.



(c) A spanning tree T_2 of G.

Figure 3.4 An example of the variant of maximum leaf spanning tree problem.

In the next, we state two theorems for the existence of such a spanning tree [2]. These theorems investigate the necessary and sufficient conditions for the existence of such a spanning tree. In the first theorem there are two conditions and in the second theorem there are three conditions.

***Theorem* 3.1.1 [2]:** *Let G = (V, E) be a connected bipartite graph with partite sets X and Y. Let* $K \leq |X|$ *be a positive integer. Then there is a spanning tree* T_G *of G with at least K leaves in X if and only if there is set* $S \subseteq X$ *such that*

(1) $| X \setminus S | \geq K$, and

(2) the induced subgraph $S \cup Y$ of G is connected.

Proof: The proof of the theorem is simple. We first consider the if part. Suppose in G there is a set S such that $| X \setminus S | \geq K$ and $(S \cup Y)$ is connected. We can easily find a spanning tree T' for the graph $(S \cup Y)$. Now suppose in T' number of leaves belonging to S is K'. It is clear that $0 \leq K' \leq |S'|$. Now since G is connected bipartite graph, for each $x \in X \setminus S$ we can add an edge (x, y)

such that $y \in Y$ to get a spanning tree T'' of G. This would mean that for all $x \in X \setminus S$, $\deg_{T''}(x) = 1$. Now since $|X \setminus S \geq K$ so, T'' would necessarily be our desired spanning tree T.

Conversely, suppose G has a spanning tree T such that number of leaves in T belonging to X is greater than or equal to K. Now let L be the set of leaves of T belonging to X. Since G is bipartite so in T, $N_G(L) \subseteq Y$. Hence if we delete the set L from T_G we still get a tree. This means that in G, $\langle (X \setminus L) \cup Y \rangle$ is connected. Now since according to the assumptions $|L| \geq K$ so, in effect we have established the existence of the set S ($=X \setminus L$) in G as defined in the conditions. Hence the result follows. □

Now we present a more stringent condition for the existence of the desired spanning tree in a bipartite graph G. The conditions are presented in the form of following theorem [2].

Theorem 3.1.2: *Let G be a connected bipartite graph with partite sets X and Y and suppose K is a positive number such that $K \leq |X|$. Then there is a spanning tree T in G such that number of leaves in T belonging to X is greater than or equal to K if and only if there is a set $S \subseteq X$ such that all of the followings hold true:*

(1) $|X \setminus S| > K$,

(2) $(S \cup Y)$ is connected, and

(3) for any subset $S' \subseteq S$, $|N_G(S')| \geq |S'| + 1$.

Proof: We first prove the only if part. Suppose there is a spanning tree T such that the number of leaves in T belonging to X is greater than or equal to K. Let $L = \{x \mid x \in X \text{ and } \deg_T(x) = 1\}$. So $|L| \geq K$. Now define S = X \ L. Now it is clear that $|X \setminus S| \geq K$. To show that S U Y is connected in G, we just need to realize that since G is bipartite so in T, $N_G(L) \subseteq Y$. Finally, since $S = X \setminus L$, we must have for all $x \in S$, $\deg_T(x) \geq 2$, because otherwise either x would be a leaf or x would have no edge implying that G is not connected, both of which are contradictions. We thus have for any set $S' \subseteq S$.

$$|S'| + 1$$

$$\leq \quad \sum (x \in S(\deg_T(x) - 1) + 1$$

$\leq \quad |N_T(S)|$

$\leq \quad |N_G(S)|$

Now we need to show the if part. Assume that the sufficient conditions are satisfied. Let $G'=S \cup Y$. G' is bipartite sets S and Y. Since G is bipartite, the assumptions about S hold in G' as well. So we must have for any subset $S' \subseteq S$ in G'.

$$|N_G(S')| \geq |S'|+1$$

$$\geq |S'|$$

Hence there is matching M from S to Y by Hall's theorem [16, 17, 18]. Now $F = (S \cup Y, M)$ is a forest where for all $x \in S$, $\deg_F(x) = 1$. Now we show that there exists a forest $\overline{F}$ such that $\deg_{\overline{F}}(x) = 1$ for all $x \in S$ as follows. We show it by an induction on the number of vertices with degree 1 in a forest. Let F' be the forest containing F such that $1 \leq \deg_{F'}(x) \leq 2$ for all $x \in S$ and let S_1 be the set of all vertices such that $\deg_{F'}(x) = 1$ for all $x \in S_1$. According to assumption we must have $N_G(S_1) \geq |S_1|$. Hence there must be an edge e joining S_1 and $Y \setminus N_{F'}(S_1)$. So in the forest $F' \cup e$, the number of vertices with degree 1 is fewer than $|S_1|$. Now we can obtain a spanning tree F' of G' when we add some edges to $\overline{F}$ between the connected components of $\overline{F}$. Finally since G is connected and bipartite, for each $x \in X \setminus S$, we can easily add an edge (x, y) of G to T' where $y \in Y$, to get the desired spanning tree T. □

Theorem 3.1.3: *Variant of maximum leaf spanning tree problem for bipartite graph is NP-Complete* [2].

.***Proof***: Consider an instance of the set-covering problem given by a collection C of subsets of the finite set $A = \bigcup_{c \in C} C$ and a positive integer $c \leq |C|$. This problem is *NP*-complete. Construct a bipartite graph $G = (X \cup Y, E)$ from the incidence relation between C and A so that X correspond to C and Y correspond to A. To ensure that G is connected, add an extra element, to the set Y and include it in every set of C. Let $k = |X| - c$.

It is easy to see that any set cover of A induces a connected subgraph in G which contains all of Y. On the other hand, any family of subsets that induces induces a connected subgraph of G which contains all of Y, is a set cover of A. Therefore, C contains a cover for A of size c or less if and only if there exists asset $S \subseteq X$ such that $|X \backslash S| \geq K$, and the induced subgraph $S \cup Y$ of G is connected. □

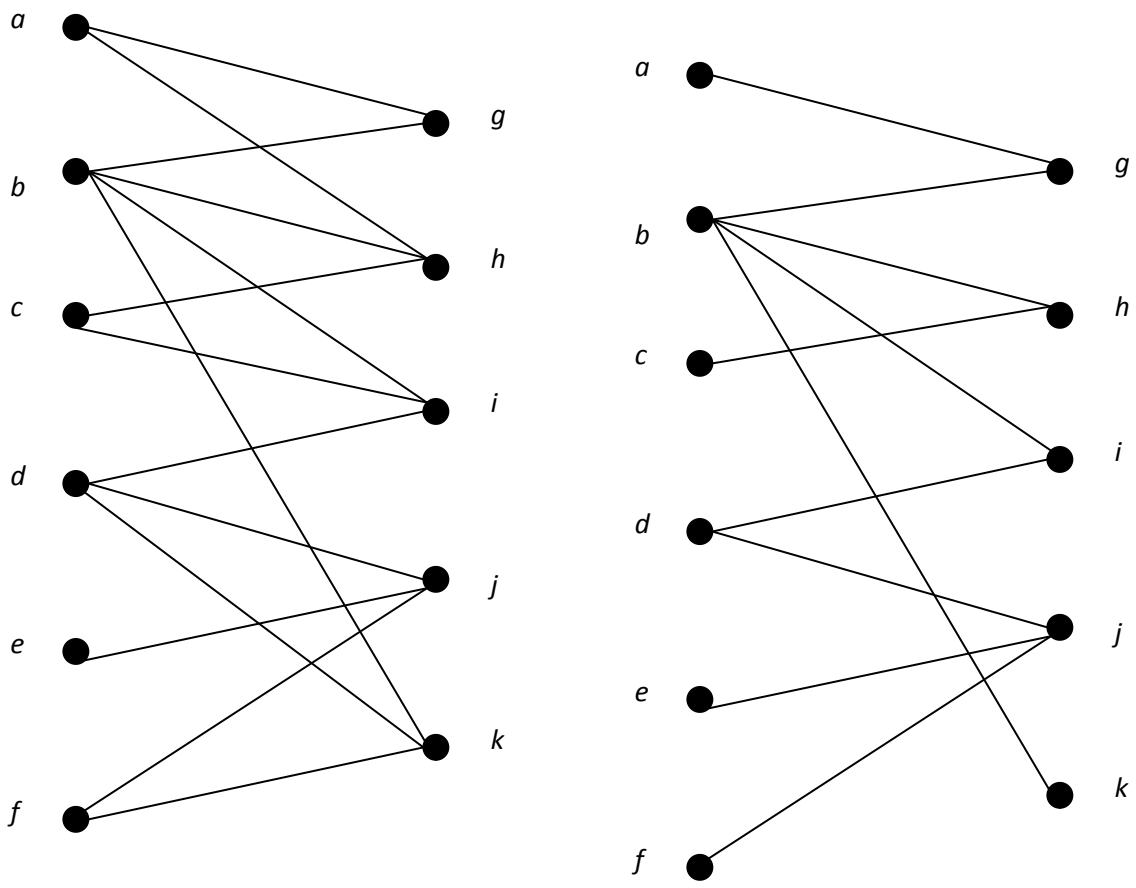
3.1.5 Exact Cover by 3-Sets (X3C)

Definition: Given a finite set X with $| X | = 3q$ and a collection C of 3-element subsets of X, we are asked the question of whether there is a sub-collection of C that partitions X.

Example: Suppose there is a set $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $C_1 = \{1, 3, 6\}$, $C_2 = \{1, 2, 3\}$, $C_3 = \{2, 4, 9\}$, $C_4 = \{2, 3, 8\}$, $C_5 = \{5, 7, 8\}$, $C_6 = \{4, 6, 9\}$. We can see that the subset C_1, C_3 and C_5 cover the set X. So C_1, C_3 and C_5 partitions the set X.

Problem 3.1.1: Given a connected bipartite graph $G = (V, E)$ of maximum degree 4 with partite sets X and Y and a positive integer $K \leq | X |$, we are asked the question whether there is a spanning tree T_G of G such that the number of leaves in T_G belonging to X is greater than or equal to K.

Example: Suppose there is a graph G with partite sets $X = \{a, b, c, d, e, f\}$ and $Y = \{g, h, i, j, k\}$. Also assume that the value of the input instance K is 3. We can find a spanning tree T_G in Figure 3.5 which has leaves belonging to X is 4. So it is our desired spanning tree.



(a) An arbitrary graph G (b) A spanning tree T_G of G

Figure 3.5 A graph with maximum degree 4 and the corresponding spanning tree.

Proof: Let (*X, C*) be an instance of the *X*3*C* problem with $|C| = p$, $|X| = 3q$, $p \geq q$, and no element of *X* occurs is more than three members of *C*. We shall now construct a bipartite graph $G = (V, E)$ with maximum degree 4 with partite sets *A* and *B*, such that (*X, C*) has a exact 3-cover if and only if *G* contains a spanning tree with at least p-q leaves in B. We begin by building a rooted tree T^* (from the bottom up) of depth $\lceil \log_2 p \rceil$ as follows:

(1) Let the numbers of *C* be the nodes at depth $d = \lceil \log_2 p \rceil$ of the tree and color each node white.

(2) Partition the nodes at depth d into sets of two nodes until it can be done and put the remaining node, if it exists, as a (singleton) member of the partition.

(3) For each member of partition, create a new node v (with color black) and join the nodes in the member to it. The newly created node will be a node at depth $d - 1$. If the node v is an even distance from a node of C, then

(a) re-color v white,

(b) adjoin a new node u to v, where the depth of u is depth(u) = depth(v)+1,

(c) color u black, and

(d) set $d = d - 1$. If $d = 0$, stop. Otherwise, go to step 2.

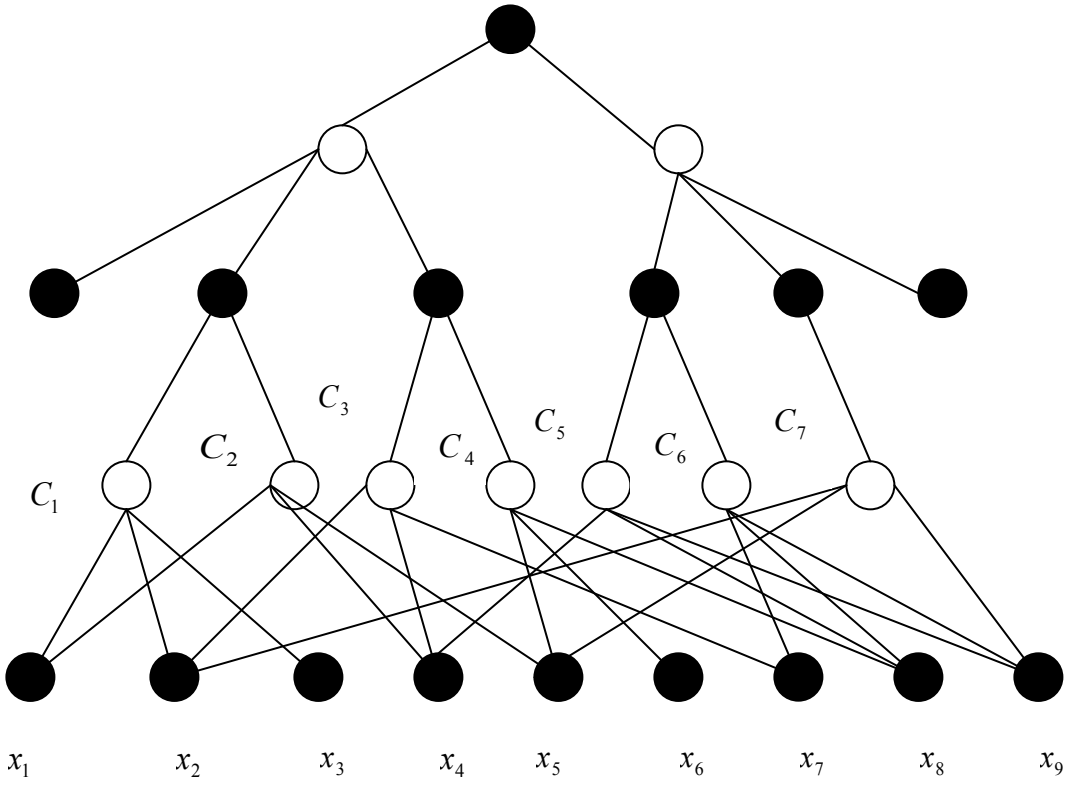


Figure 3.6 The graph constructed from an X3C instance

This concludes the construction of the tree T^*. Now create G from T^* by connecting the members of C (in T^*) and S as follows: For $c \in C$ and $x \in X$, $c \rightarrow x$ if and only if $x \in c$. Finally, color all members of X black. Taking all the black nodes as A, and all the white nodes as B along with all the edges E created by the above process, $G = (A \cup B, E)$ is clearly a bipartite graph with partite sets A and B, and maximum degree 4. It is clear that the graph G can be constructed in polynomial time. Figure 3.6 illustrates the graph that is constructed from the X3C instance (X, C) where

$X=\{ x_1, x_2, ..., x_9 \}$, and

$C_1 = \{x_1, x_2, x_3\}, C_2 = \{x_1, x_4, x_5\}, C_3 = \{x_2, x_4, x_7\}, C_4 = \{x_5, x_6, x_7\}, C_5 = \{x_4, x_8, x_9\}, C_6 = \{x_7, x_8, x_9\}, C_7 = \{x_2, x_5, x_9\}.$

The key observation here is that none of the white nodes (those in B) can be leaf nodes of a spanning unless they are members of C. This is because of the (black) leaf nodes that were introduced in step 3*b* of the construction. We will show that (X, C) has an exact 3-cover if and only if G contains a spanning tree with at least $K = p - q$ leaves in b. If there is an exact 3 cover say $\{C_1, C_2, ..., C_q\} \subset C$, then we can choose the spanning tree T_G by taking the tree T^*. In addition, for each $c = \{x, y, z\}$ in the exact cover, add the edges $\{c_i, x\}$, $\{c_i, y\}$, $\{c_i, z\}$ to T_G. T_G is clearly a spanning tree of G. Conversely, let T_G be a spanning tree of G such that B contains at least $p - q$ leaf nodes. From our observation that only the nodes in C may be leaf nodes in B, it follows that at least $p - q$ members of C are leaf nodes in T_G. It is clear that if we can show that C contains exactly $p - q$ leaf nodes, we can construct an exact 3-cover. Suppose there are at least $p - q + 1$ members of C which are leaf nodes in the spanning tree T_G. Then there are at most $q - 1$ members of C which are non-leaf nodes. These are the only nodes that connect the members of X to G. However, note that $|X| = 3q$ and each number of C. Since $(q - 1) * 3 = 3q - 3 < 3q$, this contradicts the assumption that T_G is a spanning tree. Therefore, there are exactly $p - q$ members of C that are leaf nodes.

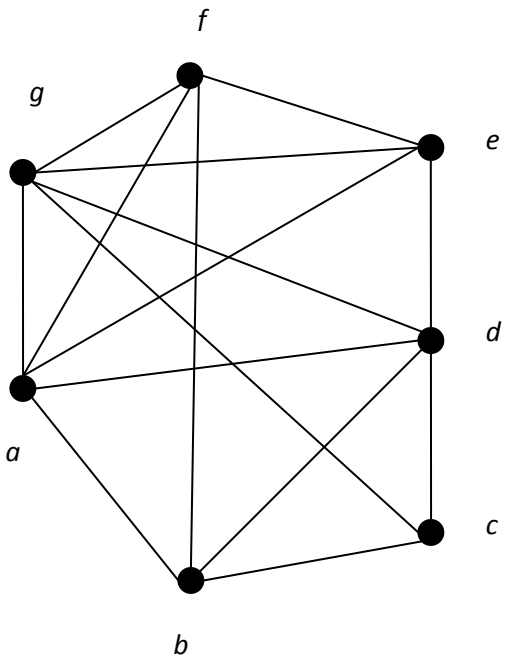
3.2 The Set Version of Maximum Leaf Spanning Tree Problems

In this section we describe the notion of "set version" of some well known decision problems. Consider the input instances of Maximum Leaf Spanning Tree Problem, where we have an integer $K \leq |V|$ as a part of the input and we are asked the question whether there exists a spanning tree T in the input graph such that T has K or more leaves. Our notion of set version would pose a similar but different problem where integer K in the input instance is replaced by a set $X \subseteq V$ [1, 2, 21]. To be specific we define the corresponding set version of the problems are describe as follows [1].

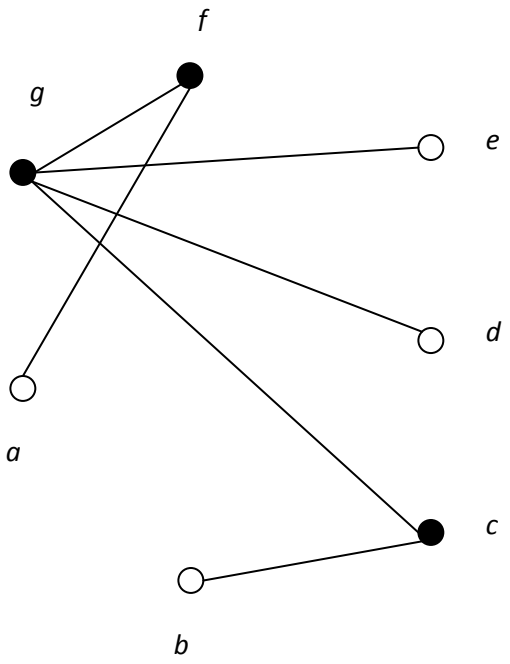
3.2.1 Set version of Maximum Leaf Spanning Tree Problem

Definition: Given a connected graph $G = (V, E)$ and $X \subseteq V$, we are asked the question whether there is a spanning tree T such that $X \subseteq \prod_T$, where $\prod_T = \{v \mid v \text{ is a leaf in } T\}$ [2, 22].

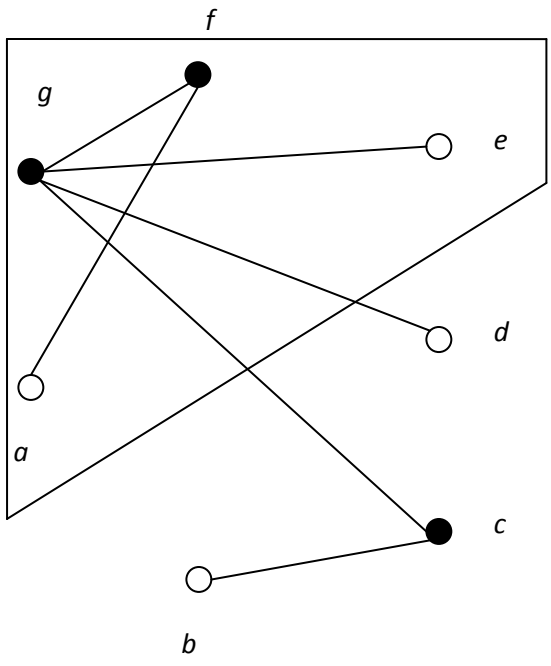
Example: Consider the graph G in Figure 3.7. Assume first that the value of K is 3. Therefore the spanning tree T_1 in Figure 3.7 (a) is a desired maximum leaf spanning tree. However, if K is 5 then T_1 fails to satisfy the requirements and hence it is not the desired spanning tree. Note carefully that even if K is 4, T_1 remains the desired spanning tree. Now in order to consider the set version of the Maximum Leaf Spanning Tree Problem, a subset X should be defined in the input instance instead of K. Suppose X is defined to be the set $\{a, f, g, e\}$. Now in T_2 in Figure 3.7(c) there are two non-leaf nodes belonging to X. So T_2 cannot be our desired spanning tree. However, if $X = \{e, d\}$ then T_3 in Figure 3.7(d) is a desired spanning tree because all of the nodes belonging to X in T_3 are leaves. Note carefully that in this case



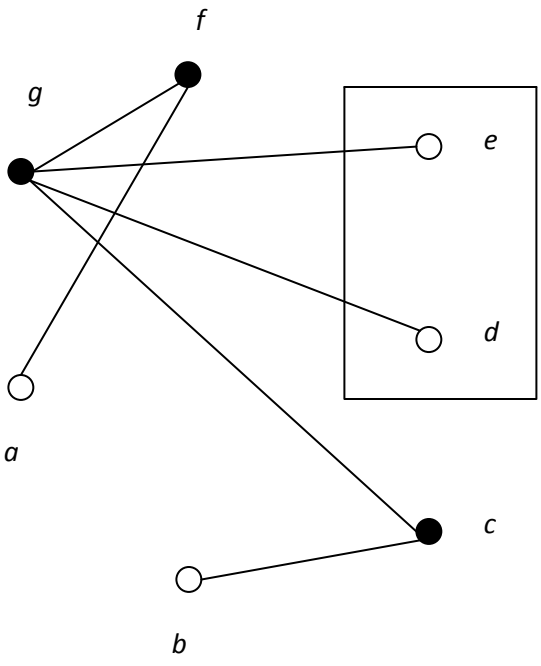
(a) An arbitrary graph of *G*.



(b) A spanning tree T_1 of *G*.



(c)A spanning tree T_2 .



(d) A spanning tree T_3 .

Figure 3.7 Set Version of Maximum Leaf Spanning Tree Problem.

We are not at all concerned about the nodes outside the given X. We are only to ensure that all the vertices belonging to X should be leaves. There may be more leaves outside X. Now we present a necessary and sufficient condition for the existence of such a spanning tree as described in problem, in the form of a theorem [18, 23]. As we point out later, the proof of theorem indicates the existence of a polynomial time algorithm to find one such spanning tree as opposed to *NP*-Completeness of the original problem.

Theorem 3.2.1 [2]***:*** *Let $G = (V, E)$ be a connected graph, $X \subseteq V$ and $Y = V \setminus X$. Then there exists a spanning tree such that $X \subseteq \prod_T$, if and only if both of the following conditions hold true:*

(1) $\{Y\}$ is connected.

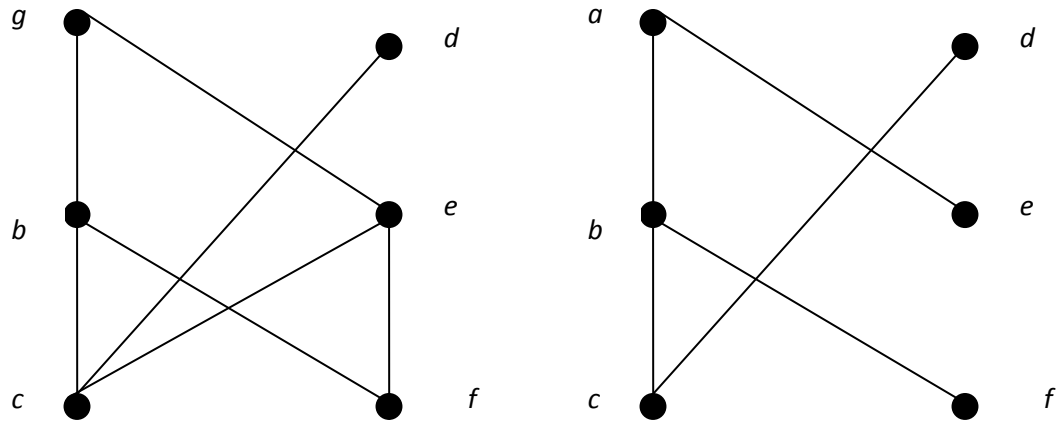
(2) Every X-node has an adjacent node in Y.

Proof: First we prove the "only if part": Suppose there is a spanning tree T in G such that $X \subseteq \prod_T$. In that case it is easy to realize that $\langle Y \rangle$ is connected since for all x $\in X$, $\deg_T$ (x) =1 and deleting X does not affect the connectivity of the rest of the graph [2, 19, 20]. The second condition is also trivially true because otherwise two X-nodes would be adjacent in T and thus one of them could not be a leaf of T.

Now we need to prove the "if part": suppose condition (1) and (2) are true. First we find a spanning tree T' for $\langle Y \rangle$, then connect each X-node to an adjacent Y-node to get a desired spanning tree T. □

Theorem 3.2.2 *Set version of the Maximum Leaf Spanning Tree Problem is polynomially solvable.*
Running Time: In the case of set version, the question about a spanning tree should be answered only for one set and we might have a polynomial time algorithm for answering the question [1].

Suppose there is a graph G with $V = \{a, b, c, d, e, f\}$ and $E = \{(a, b), (a, e), (b, c), (b, f), (c, d), (c, e), (e, f)\}$. Also assume there is a set $X = \{e, f\}$. We have to find a spanning tree T such that $X \subseteq \prod_T$. We can check it in $O(n)$ time. So the running time of the set version of the maximum leaf spanning tree is $O(n)$. Our desired spanning tree is shown in Figure 3.8.



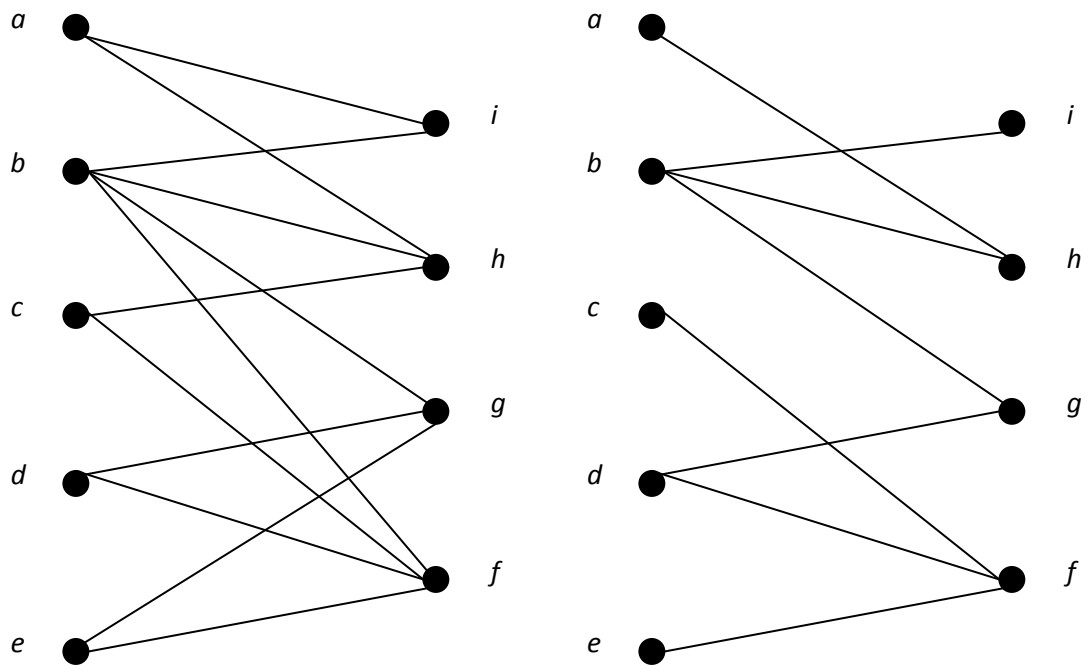
(a) An arbitrary graph *G*. (b) A spanning Tree *T*.

Figure 3.8 A graph for the illustration of set version.

3.2.2 Set Version of Variant of Maximum Leaf Spanning Tree Problem for Bipartite Graph

Definition: Let *G* be a connected bipartite graph with partite sets *X* and *Y* and $X_1 \subseteq X$. We are asked the question whether there is a spanning tree T_G such that $X_1 \subseteq \prod_T$, where $\prod_T = \{v \mid v$ is a leaf in $T\}$ [1, 2].

Example: Consider a bipartite graph *G* with partite sets $X = \{a, b, c, d, e\}$ and $Y = \{f, g, h, i\}$. Also suppose that $X_1 = \{a, c\}$. Now we find a spanning tree T_G from the bipartite graph *G*. In T_G there are three leaves, and the leaves set is $\prod_T = \{a, c, e\}$. So $X_1 \subseteq \prod_T$. Thus it is our desired spanning tree. Now we have the following theorem [1]. Figure 3.9 shows the set version of variant of maximum leaf spanning tree problem for bipartite graph.



(a) An arbitrary bipartite graph *G*. (b) A spanning tree T_G of *G*.

Figure 3.9 Set version of maximum leaf spanning tree for bipartite graph.

***Theorem* 3.2.3:** *The Set Version of the Maximum Leaf Spanning Tree problem for bipartite graph is polynomially solvable* [1, 2].

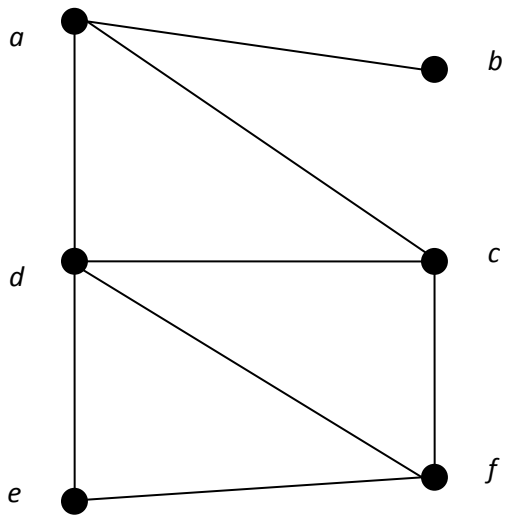
3.2.3 Set Version of the Maximum Leaf Spanning Tree Problem for Weighted Graph.

Definition: Given a connected weighted graph $G = (V, E)$, $X \subseteq V$ *and* $k > 0$, we are asked whether there is a spanning tree *T* of *G* such that $wt(T) \leq k$ and $X \subseteq \prod_T$, where $\prod_T = \{v: v$ is a leaf node of $T\}$ [1, 24].

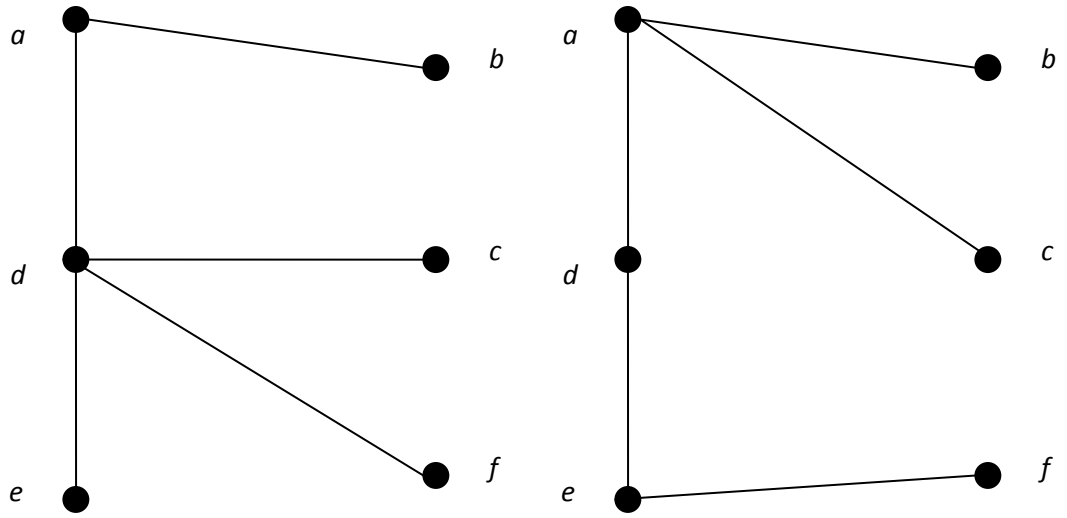
3.2.4 Set Version of Minimum Leaf Spanning Tree Problem

Definition: Given a connected graph $G = (V, E)$ and $X \subseteq V$, we are asked the question whether there is a spanning tree T_G such that $\prod_T \subseteq X$, where $\prod_T = \{v \mid v \text{ is leaf in } T\}$ [2].

Example: Consider an arbitrary graph G. Its vertex set $V = \{a, b, c, d, e, f\}$ and edge set $E = \{(a, b), (a, c), (c, d), (c, f), (d, e), (d, f), (e, f)\}$. We also assume that $X = \{a, b, c, f\}$. We find two spanning trees T_1 and T_2 of G. In T_1, the leaf set $\prod_T = \{b, c, e, f\}$. But in our spanning tree the condition $\prod_T \subseteq X$ should be meet. We see that in T_1, the condition $\prod_T \subseteq X$ does not meet. So it is not our desired spanning tree. But in T_2, the leaf set $\prod_T = \{b, c, f\}$ and it meets the condition $\prod_T \subseteq X$. So it is our desired spanning tree. Figure 3.10 shows the set version of minimum leaf spanning tree.



(a) An arbitrary graph G.



(b) A spanning tree T_1. (c) A spanning tree T_2 of *G*.

Figure 3.10 Illustration of the Set Version of Minimum Leaf Spanning Tree Problem.

CHAPTER 4

Bipartite Graph with Maximum Degree Three

The Maximum Leaf Spanning Tree problem is known to be *NP*-Complete. In [5], a variation on this problem was posed. This variation restricts the problem to bipartite graphs and asks, for a fixed integer *k,* whether or not the graph contains a spanning tree with at least *k* leaves in one of the partite sets. In [6], it was shown that not only this problem is *NP*-Complete, but that it remains *NP*-Complete for planar bipartite graphs of maximum degree four. We present an algorithm to solve this problem for bipartite graphs of maximum degree three. Here we show that this problem is quadratic time solvable for planar bipartite graphs of maximum degree three.

4.1 The Algorithm

To solve Maximum Leaf Spanning Tree Problem on a Bipartite Graph of maximum degree three first of all we find the degree $d(v)$ of all vertices. Next, we prepare two separate lists L_2 and L_3 for two degree and three degree vertices respectively. Next, for each two degree and three degree vertex we prepare three separate lists of edges. One of them contains all the edges of which the degree of both end vertices are two known as $L_{2,2}$, another one contains all the edges of which the degree of one end vertex is two and the degree of another end vertex is three know as $L_{3,2}$, and the last one contains all the edges of which the degree of both end vertices are three known as $L_{3,3}$. Next, we delete the edge from the list $L_{2,2}$ if the resulting graph remains connected, we revisited the concerned degree of vertices $d(v)$ and update all the three lists of edges accordingly. Then we delete the edge from the list $L_{3,2}$ if the resulting graph remains connected, we revisited the concerned degree of vertices $d(v)$ and update all the three lists of edges accordingly. Finally we delete the edge from the list $L_{3,3}$ if

the resulting graph remains connected, we revisited the concerned degree of vertices $d(v)$ and update all the three lists of edges accordingly. Proposed algorithm is given below:

Input*:* A bipartite graph G, with maximum degree three.

Output*:* A spanning tree T with maximum number of leaves among all possible spanning trees of G.

MAX-LEAF -SPANNING-TREE (*G*)

1. Find the degree $d(v)$ of each vertex $v \in V(G)$ and insert all two degree vertices into the list L_2 and all three degree vertices into the list L_3 ;
2. Color all edges of G to black;
3. ***for*** each vertex $u \in L_2$ ***do***
4. ***if*** color (u,v) is black and $d(v) = 2$ ***then***
5. insert edge (u,v) in $L_{2,2}$ and set color $(u,v) \leftarrow$ *green;*

end for

6. ***for*** each vertex $u \in L_3$ ***do***
7. ***if*** color (u,v) is black and $d(v) = 2$ ***then***
8. insert edge (u,v) in $L_{3,2}$ and set color $(u,v) \leftarrow$ *green;*
9. ***if*** color (u,v) is black and $d(v) = 3$ ***then***
10. insert edge (u,v) in $L_{3,3}$ and set color $(u,v) \leftarrow$ *green;*

end for

11. call EDGE-DELETE-2 (G, $L_{2,2}$);
12. call EDGE-DELETE-2-3 (G, $L_{3,2}$);
13. call EDGE-DELETE-3 (G, $L_{3,3}$);
14. **return** G

EDGE-DELETE-2(*G*, $L_{2,2}$)

1. ***for*** each edge $e=(v,v') \in L_{2,2}$ ***do***
2. delete the edge e;
3. check the connectivity of G-e;

4. ***if*** *G-e* is not connected ***then***
5. rejoin the edge *e;*
6. delete the edge e from the list of $L_{2,2}$ *;*
7. ***else***
8. delete the edge *e* from the list of $L_{2,2}$
9. *G=G-e;*
10. ***if*** *d*(*u*) = *2* for an end vertex *u* of *v* ***then***
11. delete the edge (*u,v*) from $L_{2,2}$;
12. ***if*** *d*(*u*) = 3 for an end vertex *u* of *v* ***then***
13. delete the edge (*u,v*) from $L_{3,2}$;
14. **return** *G*

EDGE-DELETE-2-3(*G*, $L_{3,2}$)

1. ***for*** each edge $e=(v,v') \in L_{3,2}$ ***do***
2. delete the edge *e*;
3. check the connectivity of *G-e;*
4. ***if*** *G-e* is not connected ***then***
5. rejoin the edge *e;*
6. delete the edge *e* from $L_{3,2}$ *;*
7. ***else***
8. delete the edge *e* from $L_{3,2}$;
9. *G = G - e;*
10. ***if*** *d*(*u*) = 3 for a neighbor *u* of *v and d*(*v*) != 1 ***then***
11. insert the edge (*u*,*v*) in $L_{3,2}$ and delete the edge (*u*,*v*) from $L_{3,3}$ *;*
12. ***if*** *d*(*u*) = *2* for a neighbor *u* of *v* and *d*(*v*) != 1 ***then***
13. insert the edge (*u*,*v*) in the head of the list $L_{3,2}$ *;*
14. **return** *G*

EDGE-DELETE-3(*G,* $L_{3,3}$)

1. ***for*** each edge $e = (v,v') \in L_{3,3}$ ***do***
2. delete the edge e;
3. check the connectivity of $G - e$;
4. ***if*** $G - e$ is not connected ***then***
5. rejoin the edge e;
6. delete e from $L_{3,3}$;
7. ***else***
8. delete the edge e from $L_{3,3}$;
9. $G = G - e$;
10. ***if*** $d(u)=3$ and $d(v)$!=1 for a neighbor u of v ***then***
11. insert the edge (u,v) in the head of the list $L_{3,3}$;
12. ***if*** $d(u')=3$ and $d(v)$!=1 for a neighbor u' of v' ***then***
13. insert the edge (u',v') in the head of the list $L_{3,3}$;
14. ***if*** $d(u) = 2$ and $d(v)$!=1 for a neighbor u of v ***then***
15. insert the edge (u,v) in the head of the list $L_{3,3}$;
16. ***if*** $d(u')=2$ and $d(v)$!=1 for a neighbor u' of v' ***then***
17. insert the edge (u',v') in the head of the list $L_{3,3}$;
18. **return** G;

4.2 Time Complexity of the Algorithm

The main result of this section is this following theorem.

Theorem 4.1 *The maximum leaf spanning tree problem on a bipartite graph of maximum degree three can be found in time* $O(n^2)$.

We prove the theorem by showing the time-complexity of each algorithm

4.2.1 Time Complexity of the Algorithm MAX-LEAF-SPANNING-TREE (G)

In Step 1, to find two degree and three degree vertices we need $O(n)$ time in worst case. In Step 2, to color all edges we need $O(n)$ time. According to Step 3 we execute Steps 4 and 5 for each vertex of degree two which is n in worst case. Individually to run Steps 4 and 5 we require constant time. As these two steps run under the loop of Step 3, steps 3-5 require $O(n)$ time in total. According to Step 6 we execute Steps 7-10 for each vertex of degree three which is n in worst case. Individually to run Steps 7-10 we require constant time. As these four steps run under the loop of Step 6, steps 7-10 require $O(n)$ time in total. Steps 11-13 need $O(n^2)$ time as shown in the following sections. Thus the overall running time of the algorithm is $O(n^2)$.

4.2.2 Time Complexity of the Algorithm EDGE-DELETE-2 (G, $L_{2,2}$)

We need to execute Steps 2 to 13 for each element of $L_{2,2.}$ In Step 2, to delete an edge we need constant time. In Step 3 we check the connectivity of the graph which requires $O(n)$ time. For all other steps (4-13) we need constant time. Thus steps 2 to 13 require $O(n^2)$ time. Thus the overall time complexity of the algorithm is $O(n^2)$.

4.2.3 Time Complexity of the Algorithm EDGE-DELETE-2-3(G, $L_{3,2}$)

We need to execute steps 2 to 13 for each element of $L_{3,2.}$ In Step 2, to delete an edge we need constant time. In Step 3 we check the connectivity of the graph which requires $O(n)$ time. For all other steps (4-13) we need constant time. Thus Steps 2 to 13 require $O(n^2)$ time. Step 14 can be done in $O(n)$ time. Thus the overall time complexity of the algorithm is $O(n^2)$.

4.2.4 Time Complexity of the Algorithm EDGE-DELETE-3(G, $L_{3,3}$)

We need to execute step 2 to 17 for each element of $L_{3,3.}$ In Step 2, to delete an edge we need constant time. In Step 3 we check the connectivity of the graph which requires $O(n)$ time. For all other steps (4-17) we need constant time. Thus Steps 2 to 17 require $O(n^2)$ time. Step 18 can be done in $O(n)$ time. Thus the overall time complexity of the algorithm is $O(n^2)$.

4.3 Correctness of the Algorithm

Theorem 4.2: *Algorithm MAX-LEAF-SPANNING-TREE* (*G*) *returns a spanning tree T, where the number of leaves in T is maximum among all possible spanning trees of the given graph G.*

We first claim the following lemma.

Lemma 1: *The algorithm returns a spanning graph.*

Proof: Let $G = (V, E)$ be the input graph and let $G' = (V',E')$ be the resulting graph. The algorithm does not add any edge or vertex; it only deletes some edges form the given input graph. So clearly $V(G') = V(G)$ and $E(G') \subseteq E(G)$. Then G' is a spanning subgraph of G. □

We then claim the Lemma 2.

Lemma 2: *The algorithm returns a spanning tree.*

Proof: Let $G = (V, E)$ be the input graph and let $G' = (V',E')$ be the resulting graph. When this algorithm deletes an edge e from the graph G, it checks the connectivity of $G-e$. If $G-e$ is not connected then the algorithm does not delete the edge e from the graph G'. So clearly G' is connected.

We know that if an edge e of a graph G belongs a cycle then e can be deleted from the graph G by remaining it connected. The algorithm deletes an edge e of G if and only if after deletion of e, the resulting graph G' remains connected. So in the final graph G' there is no edge e which creates a cycle. So clearly G' is acyclic.

As G' is connected and acyclic, G' is a tree. By Lemma 1, since G' is a spanning subgraph, G' is a spanning tree of G. □

Finally we claim the following lemma.

Lemma 3*: If the number of leaves of the resulting spanning tree obtained by the algorithm is c, then c is largest among all possible spanning trees of the input graph G.*

Proof*:* This algorithm works with each edge e of the graph G. We delete an edge e from the graph G if and only if after the deletion of the edge e, the resulting graph G' remains connected.

First of all this algorithm works for each edge *e* which both end vertices degrees are two, then works for each edge *e* which one end vertex degree is three and another end vertex degree is two finally works for each edge *e* which both end vertices degrees are three. After successful deletion of one edge *e* we revisit the degree of concerned vertices and update the above mentioned three lists of edges accordingly.

So according to this algorithm resulting spanning tree *T* has maximum number of three degree vertices. If resulting spanning tree *T* from the graph *G* which $\Delta(G) = 3$ has the maximum number of three degree vertices then *T* has maximum number of leaves. So *c* is the maximum. □

4.4 Summary

In this chapter, we present an algorithm for solving the maximum leaf spanning tree problem on a bipartite graph of maximum degree three. It is the first quadratic time algorithm for solving this problem.

CHAPTER 5

Conclusion and Future Works

In this thesis, we deal with the problem of finding maximum leaf spanning tree of a given bipartite graph of maximum degree three. We present an $O(n^2)$ time algorithm to find the maximum leaf spanning tree on a bipartite graph of maximum degree three. This is the first time that a polynomial time algorithm is proposed for solving this problem.

We now recap the key contributions of this thesis and discuss directions for future research. In Chapter 1, we give an elaborate literature review of maximum leaf spanning tree problem and also discuss our motivations on this research field. In Chapter 2, we provide some mathematical, graph-theoretic and algorithmic definitions and concepts that have been used throughout this thesis.

In Chapter 3, we define some interesting problems on spanning trees, it's variant and set version of the problems. In Chapter 4, we give the algorithm for solving the MLST problem, verify the correctness of the algorithm and determine its time complexity. The running time of the algorithm for solving the maximum leaf spanning tree problem on a bipartite graph of maximum degree three is $O(n^2)$. We believe that this maximum leaf spanning tree problem on a bipartite graph of maximum degree three outline will help different important practical applications in real world problems such as, communication networks, VLSI layout etc.

5.1 Future Scope

We first introduce an algorithm for solving the maximum leaf spanning tree problem on a bipartite graph of maximum degree three. However much remains to be done in this area and we identify the following problems on which we wish to work on in the future.

1. One can extend the algorithm presented in this thesis to find out the approximation algorithms of maximum leaf spanning tree problem on a bipartite graph of maximum degree 4 or more.
2. One can use any other algorithm to solve the maximum leaf spanning tree problem on a bipartite graph of maximum degree three, for reducing the running time of the algorithm.

List of Figures

Bibliography

[1] Rahman, M.S., Kaykobad, M., "Complexities of some interesting problems on spanning tree", Information processing Letters, Vol. 94, No. 2, pp. 93-97, 2005.

[2] Ben, P. C. Li., Toulouse, M., "Variations of maximum leaf spanning tree problem for bipartite graphs", Information processing Letters, Vol. 97, No. 4, pp. 129-132, 2006.

[3] Gray, M.R., Johnson, D.S., "Computers and Intractability", Free-man, New York, 1979.

[4] Moore, C., Robson, J.M., "Hard tiling problems with simple tiles", Discrete Computational Geometry, Vol. 4, No. 26, pp. 573-590, 2001.

[5] Storer, J. A., "Constructing full spanning trees for cubic graphs", Information Processing Letters, Vol. 13, No. 1, pp. 8-11, 1981.

[6] Lemke, P., "The maximum leaf spanning tree problem for cubic graphs is NP-Complete", IMA Publication No. 428, University of Minnesota, Vol. 428, pp. 357-363, 1988.

[7] Dijkstra, E.W., "Self-stabilizing systems in spite of distributed control", Communication of the ACM, No. 17 (1974) pp. 643 – 644, 1974.

[8] Horton, P., "On two-factors of bipartite regular graphs", Discrete Mathematics, Vol 41, No. 1, pp. 35-42, 2002.

[9] Lorys, K., Grazyna, Z., "Approximation algorithm for the maximum leaf spanning tree problem for cubic graphs", Proceedings of the 10th Annual European Symposium on Algorithms, pp. 686-698, 2002.

[10] Johnson, T., "Spanning trees with many leaves", Journal of Graph Theory", Vol. 37, pp 189-197, 2001.

[11] Sanjeev, A., Shmuel, S., "Probabilistic checking of proofs: a new characterization of NP", Journal of the ACM, Vol. 45(1998), pp. 70-122, 1998.

[12] Omer, R., "Undirected ST-connectivity in log space", STOC, pp. 376-385, 2005.

[13] Martin, F., Balaji, R., "Approximating the minimum degree spanning tree to within one from the Optimal", Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms, pp. 317-324, 1992.

[14] Lu, H., Ravi, R., "The power of local optimization: approximation algorithms for maximum leaf spanning tree", Technical Report CS 05(1996), Brown University, 1996.

[15] Bonsma, P.S., Bruggemann, T., Woeginger, G.J., "A faster fpt algorithm for finding spanning trees with many leaves", Proceedings of 28th MFCS, Vol. 2747, LNCS, pp 259–268, 2003.

[16] Kneis, J., Langer, Sen, A., Deng, H., and Guha, S., "On a graph partition problem with application to VLSI layout", IEEE International Symposium On Circuits and Systems, Vol. 5, pp. 2846-2849, 1991.

[17] Rossmanith, P., "A new algorithm for finding trees with many leaves", Proceedings of 19th ISAAC, pp. 270–281, 2008.

[18] Tetsuya, F., "The maximum-leaf spanning tree problem: formulations and facets", Willy Network, Vol. 43, No. 4, pp. 212-223, 2004.

[19] Ben, P.C.Li., Toulouse, M., "Maximum leaf spanning tree problem for grid graphs", Journal of Combinatorial Mathematics and Combinatorial Computing, Vol. 97, pp. 129-132, 2006.

[20] Clark, N., Colbourn, J., Johnson, S., "Unit disk graphs", Discrete Mathematics, Vol. 86, No. 1-3, pp. 165–177, 1990.

[21] Galbiati, G., Maffiol, F., Morzenti, A., "A short note on the approxima-bility of the maximum leaves spanning tree problem", Information Processing Letters, Vol. 52, No. 1, pp.45–49, 1994.

[22] Guha, S., Khuller, S., "Approximation algorithms for connected dominating sets", Proceedings of the Fourth Annual European Symposium on Algorithms, pp. 179–193, 1996.

[23] Solis-Oba, S., "2-approximation algorithm for finding a spanning tree with maximum number of Leaves", LNCS, Vol. 1461, pp. 441–452, 1998.

[24] Sen, A., Deng, H., and Guha, S., "On a graph partition problem with application to VLSI layout", Information Processing Letters, Vol. 43, pp. 87-94, 1992.