

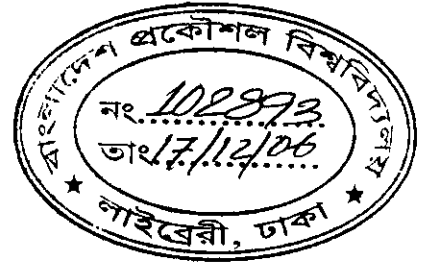
M.Sc. Engg. Thesis

An Approximation Algorithm for Edge-Ranking of Series-Parallel Graphs

by

Tanzima Hashem

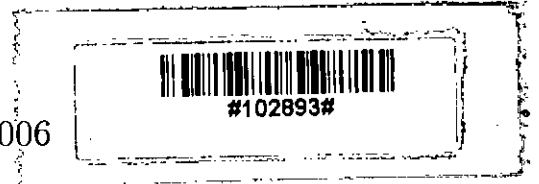
Submitted to



Department of Computer Science and Engineering
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science and Engineering


Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka 1000

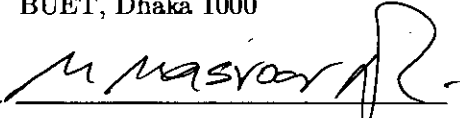
June 2006

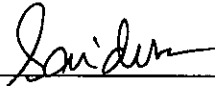


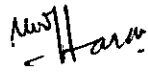
The thesis entitled "An Approximation Algorithm for Edge-Ranking of Series-Parallel Graphs", submitted by Tanzima Hashem, Roll No: 040405015P, Session: April 2004, has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Science in Computer Science and Engineering on June 10, 2006.

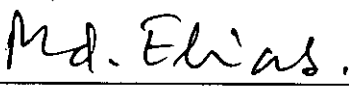
Board of Examiners

1. 

Dr. Md. Abul Kashem Mia
Professor
Department of Computer Science and Engineering
BUET, Dhaka 1000
Chairman
(Supervisor)
2. 

Dr. Muhammad Masroor Ali
Professor & Head
Department of Computer Science and Engineering
BUET, Dhaka 1000
Member
(Ex-officio)
3. 

Dr. Md. Saidur Rahman
Associate Professor
Department of Computer Science and Engineering
BUET, Dhaka 1000
Member
4. 

Dr. Masud Hasan
Assistant Professor
Department of Computer Science and Engineering
BUET, Dhaka 1000
Member
5. 

Dr. Md. Elias
Associate Professor
Department of Mathematics
BUET, Dhaka 1000
Member
(External)

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.

Tanzima

Tanzima Hashem

Candidate

Contents

<i>Board of Examiners</i>	i
<i>Candidate's Declaration</i>	ii
Acknowledgements	viii
Abstract	1
1 Introduction	2
1.1 Backgrounds	2
1.1.1 Vertex-Ranking Problem	4
1.1.2 Edge-Ranking Problem	6
1.2 Present State of the Problem	8
1.3 Scope of this Thesis	9
1.3.1 Algorithm for Finding a 2-Vertex Separator Tree of a Series-Parallel Graph	9
1.3.2 Approximation Algorithm for Edge-Ranking of a Series-Parallel Graph	10
1.3.3 Improving the Time-complexity of Vertex-Ranking Algorithm	10

1.4 Summary	10
2 Preliminaries	13
2.1 Fundamental Concepts	13
2.1.1 Graphs	13
2.1.2 Degree of a Vertex	14
2.1.3 Subgraphs	14
2.1.4 Complete Graphs and Cliques	16
2.1.5 Paths and Cycles	16
2.1.6 Connected Components and Separators	17
2.1.7 Trees	18
2.1.8 Partial k -Trees	19
2.1.9 Tree-Decomposition	19
2.1.10 Separator Tree	20
2.2 Series-Parallel Graphs	21
2.2.1 Binary Decomposition Tree	22
2.3 Complexity of Algorithms	24
2.3.1 Complexity Classes: \mathcal{P} and \mathcal{NP}	25
2.3.2 \mathcal{NP} -Complete Problem	25
2.4 Approximation Algorithm and Approximation Ratio	26
3 2-Vertex-Separator Tree	28
3.1 Preliminaries	28

CONTENTS

3.2	The Algorithm	34
3.3	Conclusion	44
4	Approximation Algorithm	45
4.1	The Algorithm	45
4.1.1	An Example	49
4.2	Approximation Ratio	50
4.2.1	Deviation from Optimality	53
4.3	Conclusion	54
5	Conclusion	56

List of Figures

1.1	A graph G .	3
1.2	A minimum vertex-coloring of graph G .	4
1.3	A minimum edge-coloring of graph G .	4
1.4	An optimal vertex-ranking of graph G .	5
1.5	An optimal edge-ranking of graph G .	7
2.1	A graph G .	14
2.2	Subgraphs induced by vertices and edges of G .	15
2.3	A complete graph and clique.	16
2.4	Separators of a graph.	17
2.5	A tree T .	18
2.6	(a) A graph G , and (b) its tree-decomposition.	20
2.7	Separator trees of a graph.	21
2.8	Series and parallel connection of a series-parallel graph.	23
2.9	A series-parallel graph G .	23
2.10	Binary decomposition tree of a series-parallel graph.	24
3.1	A series-parallel graph G is decomposed by removing a vertex.	29

3.2	A series-parallel graph G is decomposed by removing two vertices. . .	30
3.3	Components after removing u , where G_2 composed from G_3 and G_4 . . .	31
3.4	Components after removing u , where G_2 composed from only G_3 . . .	32
3.5	Components after removing u and v	33
3.6	A 2-vertex-separator tree and a binary decomposition tree of G	35
4.1	Graphs associated with the nodes of a 2-vertex-separator tree.	46
4.2	Steps of SP_Approx_Rank.	49
4.3	1-edge-separator tree of G having series connection.	51
4.4	1-edge-separator tree of G	52
4.5	The optimal edge-ranking of a series-parallel graph	54
4.6	The approximate edge-ranking of a series-parallel graph	55

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 my thesis. His in-depth knowledge in Graph Theory and his logical way of thinking have been very helpful for the successful completion of this work. I am deeply grateful to him for his cooperation.

I would like to thank the members of my thesis committee for their patience in understanding my work. I warmly thank Professor Dr. Muhammad Masroor Ali, Dr. Md. Saidur Rahman, Dr. Masud Hasan and Dr. Md. Elias for their valuable suggestions.

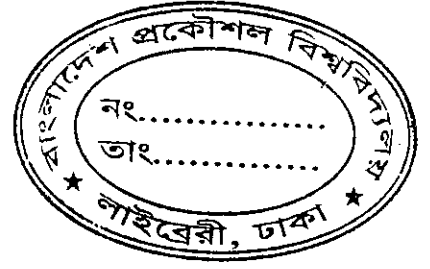
I am thankful to all of my teachers, colleagues, and friends for their support during the whole period of my thesis.

Finally, I owe my loving thanks to my husband, parents, brothers, and sisters. Without their encouragement it would have been impossible for me to finish this work.

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

Abstract

This thesis deals with an approximation algorithm for finding edge-rankings of series-parallel graphs. An edge-ranking of a graph G is a labeling of its edges with positive integers such that every path between two edges with the same label i contains an intermediate edge with label $j > i$. An edge-ranking is optimal if the least number of distinct labels among all possible edge-rankings are used by it. The edge-ranking problem is to find an optimal edge-ranking of a given graph. Analogously, the vertex-ranking problem can be defined. The edge-ranking problem of graphs has important applications like scheduling the parallel assembly of a complex multi-part product from its components and parallel computation. The edge-ranking problem is NP-complete for series-parallel graphs, that is, finding a polynomial-time algorithm for solving the edge-ranking problem on series-parallel graphs with unbounded maximum degree is unlikely. In this thesis, we present a linear-time algorithm for finding a 2-vertex-separator tree of a series-parallel graph G and a linear-time approximation algorithm for finding the edge-ranking of a given series-parallel graph G using the 2-vertex-separator tree of G . Obtaining the 2-vertex-separator tree of G immediately improves the running time of the known best algorithm that finds an optimal vertex-ranking of a series-parallel graph.



Chapter 1

Introduction

In this chapter, we provide the necessary background, present state and motivation for this study on the rankings of graphs, define the problem and scope of this thesis. In Section 1.1, we discuss the historical background on graph coloring. We also define the vertex-ranking and the edge-ranking problem, related applications and review the results on the ranking of graphs. Section 1.2 represents the present state of the problem and Section 1.3 deals with the scope of this thesis. At last, in Section 1.4, we discuss the results obtained for solving the problems of this thesis and compare our results with the previously achieved ones.

1.1 Backgrounds

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 theory.

Graph coloring theory not only plays an important role in discrete mathematics,

but also is of interest for its applications. Graph coloring deals with the fundamental problem of partitioning a set of objects into classes according to certain rules. A graph $G = (V, E)$ with n vertices and m edges consists of a vertex set $V = \{v_1, v_2, \dots, v_n\}$ and an edge set $E = \{e_1, e_2, \dots, e_m\}$, where an edge in E joins two vertices in V . Figure 1.1 depicts a graph of seven vertices and nine edges, where vertices are drawn by circles, edges by lines, vertex names next to the circles and edge names next to the lines.

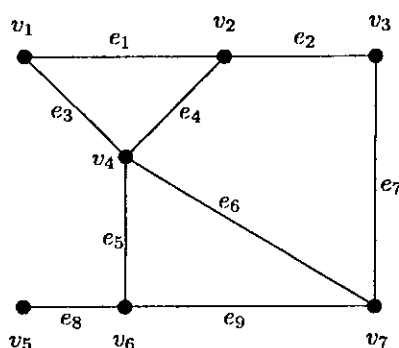
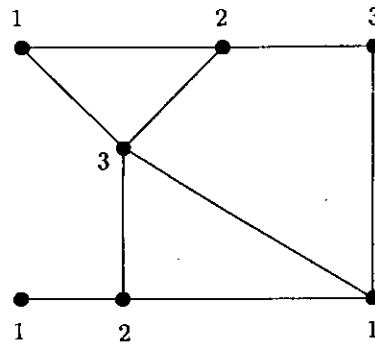
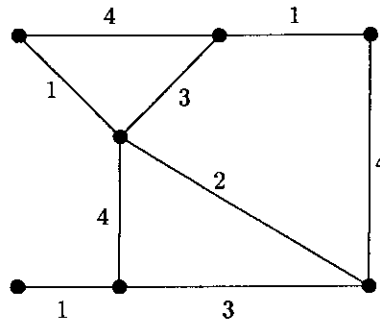


Figure 1.1: A graph G .

The vertex-coloring problem and the edge-coloring problem are two of the fundamental problems on graphs. The *vertex-coloring problem* is to color the vertices of a given graph with the minimum number of colors so that no two adjacent vertices are assigned the same color. Figure 1.2 depicts a minimum vertex-coloring of a graph G using three colors, where colors are drawn next to the vertices. The *edge-coloring problem* is to color the edges of a given graph with the minimum number of colors so that no two adjacent edges are assigned the same color. Figure 1.3 depicts a minimum edge-coloring of G using four colors, where colors are drawn next to the edges. The vertex-ranking problem and the edge-ranking problem are restrictions of the vertex-coloring problem and the edge-coloring problem, respectively.

Figure 1.2: A minimum vertex-coloring of graph G .Figure 1.3: A minimum edge-coloring of graph G .

1.1.1 Vertex-Ranking Problem

A *vertex-ranking* of a graph G is a labeling (ranking) of the vertices of G with positive integers such that every path between any two vertices with the same label i contains a vertex with label $j > i$ [9]. Clearly a vertex-labeling is a vertex-ranking if and only if, for any label i , deletion of all vertices with labels $> i$ leaves connected components, each having at most one vertex with label i . The integer label of a vertex is called the *rank* of the vertex. The minimum number of ranks needed for a vertex-ranking of G is called the *vertex-ranking number* of G and is denoted by $r(G)$. A vertex-ranking of G using the minimum number of ranks is called an *optimal vertex-ranking* of G . The *vertex-ranking problem* is to find an optimal vertex-ranking

ranking problem for trees [9], where n is the number of vertices of the input tree. Then Schäffer obtained a linear-time algorithm by refining their algorithm and its analysis [23]. Deogun *et al.* gave algorithms to solve the vertex-ranking problem for interval graphs in $O(n^3)$ time and for permutation graphs in $O(n^6)$ time [6]. Bodlaender *et al.* presented a polynomial-time sequential algorithm to solve the vertex-ranking problem for partial k -trees, that is, graphs of treewidth bounded by a fixed integer k [4]. Kloks *et al.* have presented an algorithm for computing the vertex-ranking number of an asteroidal triple-free graph in time polynomial in the number of vertices and the number of minimal separators [16]. Newton and Kasbem presented an efficient optimal algorithm for vertex-ranking of permutation graphs in $O(n^3)$ time [22]. Sun-yuan Hsieh solved the vertex ranking problem of a starlike graph in $O(n)$ time [8].

1.1.2 Edge-Ranking Problem

The edge-ranking problem is defined analogously as for the vertex-ranking problem. An *edge-ranking* of a graph G is a labeling of the edges of G with positive integers such that every path between two edges with the same label i contains an edge with label $j > i$ [11, 7]. Clearly an edge-labeling is an edge-ranking if and only if, for any label i , deletion of all edges with labels $> i$ leaves connected components, each having at most one edge with label i . The minimum number of ranks needed for an edge-ranking of G is called the *edge-ranking number* of G and is denoted by $r'(G)$. An edge-ranking of G using the minimum number of ranks is called an *optimal edge-ranking* of G . The *edge-ranking problem* is to find an optimal edge-ranking of a given graph. The constraints for the edge-ranking problem imply that two adjacent edges cannot have the same rank. Thus the edge-ranking problem is a restriction of the edge-coloring problem. Figure 1.5 depicts an optimal edge-ranking of a graph

using six ranks, where ranks are drawn next to the edges. The problem of finding

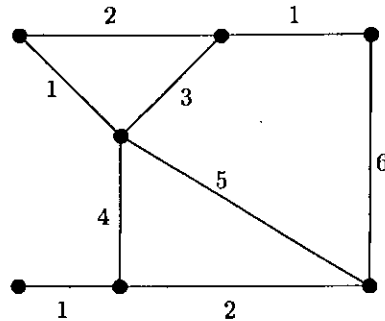


Figure 1.5: An optimal edge-ranking of graph G .

an optimal edge-ranking of a graph G has applications in scheduling the parallel assembly of a complex multi-part product from its components. The edge-ranking problem for a graph G is also equivalent to finding an edge-separator tree of G having the minimum height. An edge-separator tree with minimum height corresponds to a parallel computation scheme having the minimum computation time [21].

We next review the results on the edge-ranking problem. The problem of finding an optimal edge-ranking was first studied by Iyer *et al.* in 1991 as they found that the problem has an application in scheduling the parallel assembly of multipart products. They gave an $O(n \log n)$ time approximation algorithm for finding an edge-ranking of trees T using at most twice the minimum number of ranks, where n is the number of vertices in T [11]. Their approximation algorithm uses the vertex-ranking algorithm in [9] as a subroutine. The main open problem in their paper is to determine whether the edge-ranking problem is in \mathcal{P} , or if it is \mathcal{NP} -hard. Later de la Torre *et al.* have given an exact algorithm to solve the edge-ranking problem for trees in time $O(n^3 \log n)$ by means of a two-layered greedy method [26]. Thus the edge-ranking problem when restricted to trees is in \mathcal{P} . However, Lam and Yue have proved that the edge-ranking problem is \mathcal{NP} -hard for graphs in general [17].

and they have solved the optimal edge-ranking problem on trees in linear-time [18].

A natural generalization of an ordinary edge-ranking is the *c-edge-ranking* [27]. A *c-edge-ranking* of a graph G , for a positive integer c , is a labeling of the edges of G with integers such that, for any label i , deletion of all edges with labels $> i$ leaves connected components, each having at most c edges with label i . Clearly an ordinary edge-ranking is a 1-edge-ranking. The minimum number of ranks needed for a c -edge-ranking of G is called the *c-edge-ranking number*, and is denoted by $r'_c(G)$. A c -edge-ranking of G using $r'_c(G)$ ranks is called an *optimal c-edge-ranking* of G . The *c-edge-ranking problem* is to find an optimal c -edge-ranking of a given graph G . Zhou *et al.* gave an algorithm to find an optimal c -edge-ranking of a given tree T for any positive integer c in time $O(n^2 \log \Delta)$, where Δ is the maximum vertex-degree of T [27]. Kashem *et al.* gave a polynomial time sequential algorithm for generalized edge-ranking of partial k -trees with bounded maximum degree [13].

1.2 Present State of the Problem

In graph theory, series-parallel graphs related algorithms have been intensively studied in recent years. But some interesting problems like edge-ranking in this domain are \mathcal{NP} -complete and thus near optimal polynomial-time solution is required. A polynomial-time algorithm to solve the generalized edge-ranking problem on partial k -trees with bounded maximum degree has been given by Kashem *et al.* [13]. Since a series-parallel graph is a partial 2-tree, a polynomial-time algorithm for series-parallel graphs with bounded maximum degree is immediately yielded by their algorithm. However, the edge-ranking problem is \mathcal{NP} -complete for general series-parallel graphs [12], that is, finding a polynomial-time algorithm for solving the edge-ranking problem on series-parallel graphs with unbounded maximum degree is unlikely. Therefore it is necessary to design a polynomial-time

approximation algorithm for edge-ranking of general series-parallel graphs which will find a near optimal solution. There is still no approximation algorithm for edge-ranking of general series-parallel graphs.

1.3 Scope of this Thesis

We summarize our developed and improved algorithms for series-parallel graphs in this thesis.

1.3.1 Algorithm for Finding a 2-Vertex Separator Tree of a Series-Parallel Graph

Since a series-parallel graph is a partial 2-tree, it has a 3-vertex-separator tree [14]. We first prove that a series-parallel graph has a 2-vertex-separator tree. Consider the process of starting with a connected graph G and partitioning it recursively by deleting at most 2 vertices from each of the remaining connected components until the graph becomes empty. The tree representing the recursive decomposition is called 2-vertex-separator tree. To prove that a series-parallel graph has a 2-vertex-separator tree, at first, we show that a (connected) series-parallel graph can be disconnected by removing at most two vertices. However, disconnected components that do not have the series-parallel structure may be yielded by this process. So we also show that every such component has at least one cut-vertex. This immediately proves that a series-parallel graph has a 2-vertex-separator tree. Then based on this proof and using binary decomposition tree of a series-parallel graph G we present a linear-time algorithm for constructing a 2-vertex-separator tree of G .

1.3.2 Approximation Algorithm for Edge-Ranking of a Series-Parallel Graph

We present a linear-time approximation algorithm using the 2-vertex-separator tree for finding the edge-ranking of a series-parallel graph. Solving the edge-ranking problem is equivalent to finding the minimum height 1-edge-separator tree. The problems on series-parallel graphs are generally solved using binary decomposition tree. But in this thesis we first construct a 2-vertex-separator tree using binary decomposition tree and then using the 2-vertex-separator we tree find the edge-ranking of a series-parallel graph. We also calculate the approximation ratio of the algorithm.

1.3.3 Improving the Time-complexity of Vertex-Ranking Algorithm

Obtaining the 2-vertex-separator tree immediately improves the upper bound of the optimal vertex-ranking number and thereby running time of the known best algorithm that finds the optimal vertex-ranking of a series-parallel graph. Kashem *et al.* give the algorithm for solving vertex-ranking problem of order $O(n^7 \log_2^7 n)$ using 3-vertex-separator tree [15]. If we use 2-vertex-separator tree, the running time improves to $O(n^5 \log_2^7 n)$.

1.4 Summary

The known results of algorithms for solving the edge-ranking problem on different types of graphs are summarized in Table 1.1. The main result of this thesis can be divided into two parts: a linear-time algorithm for constructing a 2-vertex-

Graphs	Time	Value of c	References
Trees	$O(n)$	$c = 1$	[18]
Trees	$O(n^2 \log \Delta)$	any positive integer	[27]
Partial k -trees with bounded degrees	effectively $n^{O(\Delta k^2)}$	any positive integer	[13]
Series-parallel graph with bounded degrees	$O(n^{18\Delta+2}(\Delta \log_2 n)^8)$	$c = 1$	[13]
Series-parallel graph (unbounded degrees)	\mathcal{NP} -Complete	$c = 1$	[12]

Table 1.1: Algorithms for edge-ranking.

separator tree of a series-parallel graph, and a linear-time approximation algorithm for finding an edge-ranking of a series-parallel graph using 2-vertex-separator tree with an approximation ratio of $2\Delta(h+1)/\log_2 n$, where Δ is the maximum vertex degree of a series-parallel graph G , h is the height of the 2-vertex-separator tree and n is the number of vertices in G . Besides these, we improve the running time of the known best algorithm for solving the vertex-ranking problem of a series-parallel graph.

The thesis is organized as follows. Chapter 2 gives preliminary definitions and representation of series-parallel graphs. Chapter 3 gives a linear-time algorithm for constructing a 2-vertex-separator tree of a series-parallel graph. Chapter 4 presents a linear-time approximation algorithm for edge-ranking of a series-parallel graph using

the 2-vertex-separator tree with an approximation ratio of $2\Delta(h+1)/\log_2 n$. Chapter 5 concludes with a discussion of the improved algorithm for solving the vertex-ranking problem on series-parallel graphs, the results of the proposed algorithm and future works.

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 graph which are related to and used through out this thesis. Section 2.2 defines a special type of graph, series-parallel graph. It also introduces different properties of a series-parallel graph and representation of series-parallel graph through the binary decomposition tree. Section 2.3 discusses complexity classes of the algorithm. Finally in Section 2.4 we define approximation algorithm and the approximation ratio.

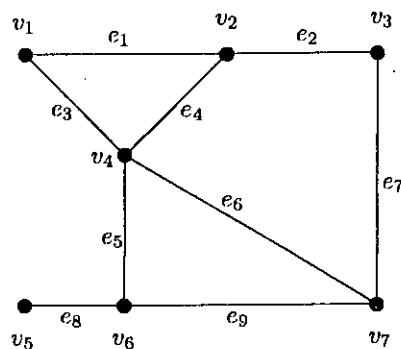
2.1 Fundamental Concepts

2.1.1 Graphs

Let $G = (V, E)$ be a graph. We call $V(G)$ or V the vertex-set of the graph G , and $E(G)$ or E the edge-set of G . If $e = (v, w)$ is an edge, then e is said to join the vertices v and w , and these vertices are then said to be *adjacent*. In this case we



also say that w is a *neighbor* of v , and that e is *incident* to v and w . A *loop* is an edge whose endpoints are equal. *Parallel edges* or *multiple edges* are edges that have the same pair of endpoints. A *simple graph* is a graph having no loops or multiple edges. The graph in which loops and multiple edges are allowed is called a *multigraph*. Sometimes a simple graph is simply called by a graph only if there is no danger of confusion. A graph is *finite* if its vertex set and edge set are finite. Every graph mentioned in this thesis is finite.

Figure 2.1: A graph G .

2.1.2 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 $d(v_1)$ v_1 is 2 and the maximum degree Δ of G , is 4 as $d(v_4)$ is 4.

2.1.3 Subgraphs

A *subgraph* of a graph $G = (V, E)$ is a graph $H = (V_H, E_H)$ such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$, we write $H \subseteq G$ and say that G contains H . If H contains

all the edges of G that join two vertices in V_H , then H is said to be the *subgraph induced by V_H* , and is denoted by $G[V_H]$. If V_H consists of exactly the vertices on which edges in E_H are incident, then H is said to be the *subgraph induced by E_H* , and is denoted by $G[E_H]$. Figure 2.2(a) depicts a subgraph of G in Figure 2.1 induced by

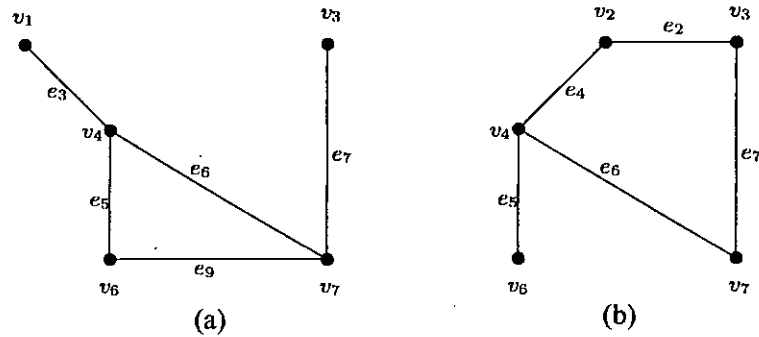


Figure 2.2: (a) A subgraph induced by $\{v_1, v_3, v_4, v_6, v_7\}$ of G in Figure 2.1, and (b) a subgraph induced by $\{e_2, e_4, e_5, e_6, e_7\}$ of G .

$\{v_1, v_3, v_4, v_6, v_7\}$ and Figure 2.2(b) depicts a subgraph induced by $\{e_2, e_4, e_5, e_6, e_7\}$.

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 the edges incident to v . More generally, if V' is a subset of V , then $G - V'$ is the subgraph of G obtained by deleting the vertices in V' and all the edges incident to them. Then $G - V'$ is a subgraph of G induced by $V - V'$. 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' .

2.1.4 Complete Graphs and Cliques

A *complete graph* is a simple graph in which every pair of vertices has an edge. A *clique* is a set of pairwise adjacent vertices in a graph. A complete graph has many subgraphs that are not cliques, but every induced subgraph of a complete graph is a clique. Figure 2.3(a) is both a complete graph and a clique with six vertices. Subgraph with $\{v_1, v_2, v_3, v_4\}$ in Figure 2.3(b) is a clique.

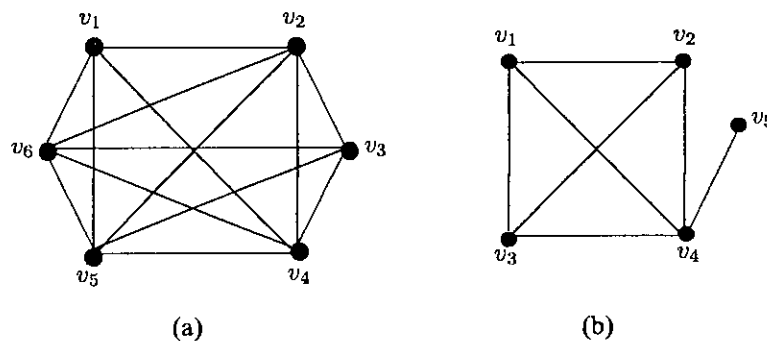


Figure 2.3: (a) A complete graph, and (b) subgraph with $\{v_1, v_2, v_3, v_4\}$ is a clique.

2.1.5 Paths and Cycles

A *walk* of length k is a sequence $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$ of vertices and edges such that $e_i = v_{i-1}, v_i$ for all $i, 1 \leq i \leq k$. A *trail* is a walk with no repeated edge. A *path* is walk with no repeated vertex. A u, v -walk has first vertex u and last vertex v . These two vertices u and v are endpoints of the u, v -walk. Normally, the path is denoted by the sequence of vertices $v_0, v_1, v_2, \dots, v_k$. The length of the path is calculated by the number of vertices less one. A walk is closed if it has length at least one and its endpoints are equal. A *cycle* is a closed trail in which “first = last” is the only vertex repetition. In Figure 2.1, an example of a path forming no cycle is $v_5v_6v_4v_2v_3$ from v_5 to v_3 and an example of cycle is $v_6v_7v_4v_6$.

2.1.6 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*. A graph which is not connected is called a *disconnected graph*. Separation of a graph can be done in two ways: using vertex separator and edge separator. Separator disconnects a graph into more than one components. A *vertex separator*

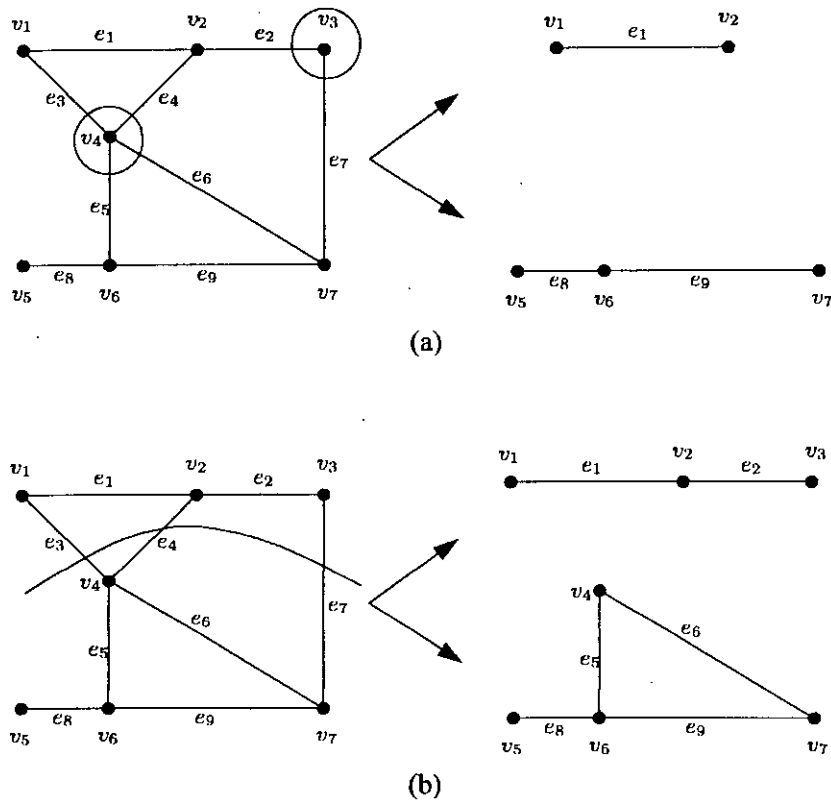


Figure 2.4: Separation of a graph G (a) with a vertex separator, and (b) an edge separator.

of a connected graph G is a set of vertices whose deletion disconnects G . The graph G in Figure 2.4(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 . The graph G in Figure 2.4(b) has

an edge separator $\{e_3, e_4, e_7\}$.

2.1.7 Trees

A graph having no cycle is *acyclic*. A *forest* is an acyclic graph; a *tree* is a connected acyclic graph. The vertices in a tree are usually called *nodes*. 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 a tree is generally drawn at the top. Figure 2.5 shows an example of a tree T , where v_1 is the root of T . Every node u

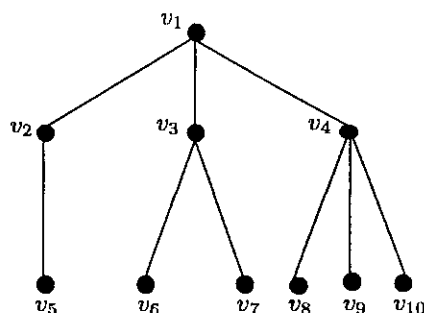


Figure 2.5: A tree T .

other than the root is connected by an edge to some other node p called the *parent* of u . We also call u a *child* of p . We draw the parent of a node above that node. For example, in Figure 2.5, v_1 is the parent of v_2 , v_3 and v_4 , while v_3 is the parent of v_6 and v_7 ; on the other hand v_2 , v_3 and v_4 are children of v_1 , while v_6 and v_7 are children of v_3 . A *leaf* is a node of a tree that has no children. That is a leaf is a vertex of degree 1. An *internal node* is a node that has one or more children. Thus every node of a tree is either a leaf or an internal node, but not both. A *binary tree* is the tree where each node does not have more than two children.

In a tree T , a node u together with all of its proper descendants, if any, is called a *subtree* of T . Node u is the root of this subtree. Referring again to Figure 2.5,

nodes v_3 , v_6 and v_7 form a subtree, with root v_3 . Finally, the entire tree in Figure 2.5 is a subtree of itself, with root v_1 . The *height of a node u* in a tree is the length of a longest path from u to a leaf. The *height of a tree* is the height of the root. The *depth of a node u* in a tree is the length of a path from the root to u . The *level of a node u* in a tree is the height of the tree minus the depth of u . In Figure 2.5, for example, node v_3 is of height 1, depth 1 and level 1. The tree in Figure 2.5 has height 2.

2.1.8 Partial k -Trees

A natural generalization of ordinary trees is the so-called *k -trees*. The class of k -trees is defined recursively as follows [3]:

- (a) A complete graph with k vertices is a k -tree.
- (b) If $G = (V, E)$ is a k -tree and k vertices v_1, v_2, \dots, v_k induce a complete subgraph of G , then $G' = (V \cup \{w\}, E \cup \{(v_i, w) \mid 1 \leq i \leq k\})$ is a k -tree, where w is a new vertex not contained in G .
- (c) All k -trees can be formed with rules (a) and (b).

A graph is called a *partial k -tree* if it is a subgraph of a k -tree.

2.1.9 Tree-Decomposition

A *tree-decomposition* of a graph $G = (V, E)$ is a pair (T, S) , where $T = (V_T, E_T)$ is a tree and $S = \{X_x \mid x \in V_T\}$ is a collection of subsets of V satisfying the following three conditions [17]:

- (a) $\bigcup_{x \in V_T} X_x = V$;
- (b) for every edge $e = (v, w) \in E$, there exists a node $x \in V_T$ with $v, w \in X_x$;

and

- (c) for all $x, y, z \in V_T$, if node y lies on the path from node x to node z in T , then $X_x \cap X_z \subseteq X_y$.

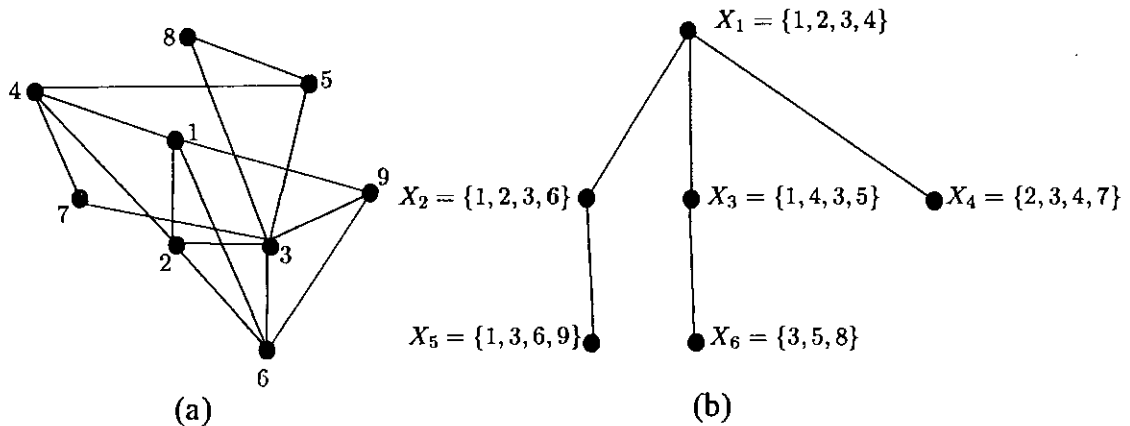


Figure 2.6: (a) A graph G , and (b) its tree-decomposition.

The width of a tree-decomposition (T, S) is $\max_{x \in V_T} |X_x| - 1$. The *tree width* of a graph G is the minimum width of a tree-decomposition of G , taken over all possible tree-decompositions of G . The width of the tree-decomposition shown in Figure 2.6(b) of the graph G of Figure 2.6(a) is 3. A graph G with treewidth $\leq k$ is called a partial k -tree. Every partial k -tree G has a tree-decomposition (T, S) with treewidth $\leq k$ and $n_T \leq n$, where n_T is the number of nodes in T [14]. So every node of tree-decomposition (T, S) of a partial k -tree can contain maximum $(k + 1)$ vertices. So it immediately implies partial k -tree has a $k + 1$ -vertex-separator tree.

2.1.10 Separator Tree

There are two types of separator trees: vertex-separator tree and edge-separator tree. Consider the process of starting with a connected graph G and partitioning

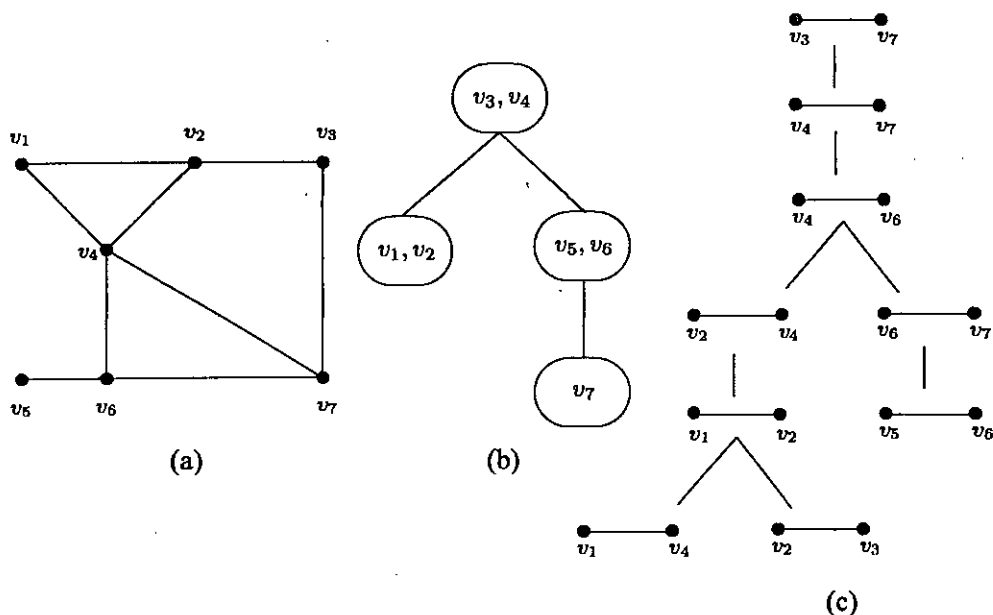


Figure 2.7: (a) A graph G , (b) its 2-vertex-separator tree, and (c) its 1-edge-separator tree.

it recursively by deleting at most c vertices from each of the remaining connected components until the graph becomes empty. The tree representing the recursive decomposition is called a c -vertex-separator tree of G . Analogously we can define c -edge-separator tree of G . Figure 2.7(b) illustrates a 2-vertex-separator tree of the graph G depicted in Figure 2.7(a), where the vertex names of deleted ones are drawn in ovals. Again Figure 2.7(c) illustrates a 1-edge-separator tree of the graph G depicted in Figure 2.7(a).

2.2 Series-Parallel Graphs

Now we will introduce a very special kind of graph known as *series-parallel graph* which is very similar to series-parallel circuit. A series-parallel graph is defined

recursively as follows.

- (1) A graph G of a single edge is a series-parallel graph. The end points s and t of the edge are called the terminals of G .
- (2) Let G_1 be a series-parallel graph with terminals s_1 and t_1 , and let G_2 be another series-parallel graph with terminals s_2 and t_2 .
 - (a) A graph G obtained from G_1 and G_2 by identifying vertex t_1 with vertex s_2 is a series-parallel graph whose terminals are $s = s_1$ and $t = t_2$. Such a connection is called a *series connection*, and G is denoted by $G = G_1 \bullet G_2$. (See Figure 2.8(a).)
 - (b) A graph G obtained from G_1 and G_2 by identifying s_1 with s_2 and t_1 with t_2 is a series-parallel graph whose terminals are $s = s_1 = s_2$ and $t = t_1 = t_2$. Such a connection is called a *parallel connection*, and G is denoted by $G = G_1 \parallel G_2$. (See Figure 2.8(b).)

A series-parallel graph is a partial 2-tree. So it has a tree-decomposition which implies series-parallel graph has a 3-vertex-separator tree. Another property of a series-parallel graph is its number of edges. A series-parallel graph on n vertices has at most $2n - 3$ edges [2].

2.2.1 Binary Decomposition Tree

The construction of a series-parallel graph can be represented by a *binary decomposition tree* T_b [25]. Every internal node of T_b is either a *s*-node or a *p*-node and every leaf node of T_b represents a subgraph of G induced by two vertices s and t connected by the edge (s, t) . Figure 2.9 illustrates a series-parallel graph G and Figure 2.10 illustrates its binary decomposition tree T_b . Labels *s* and *p* attached to internal nodes in T_b indicate series and parallel connections, respectively, and nodes

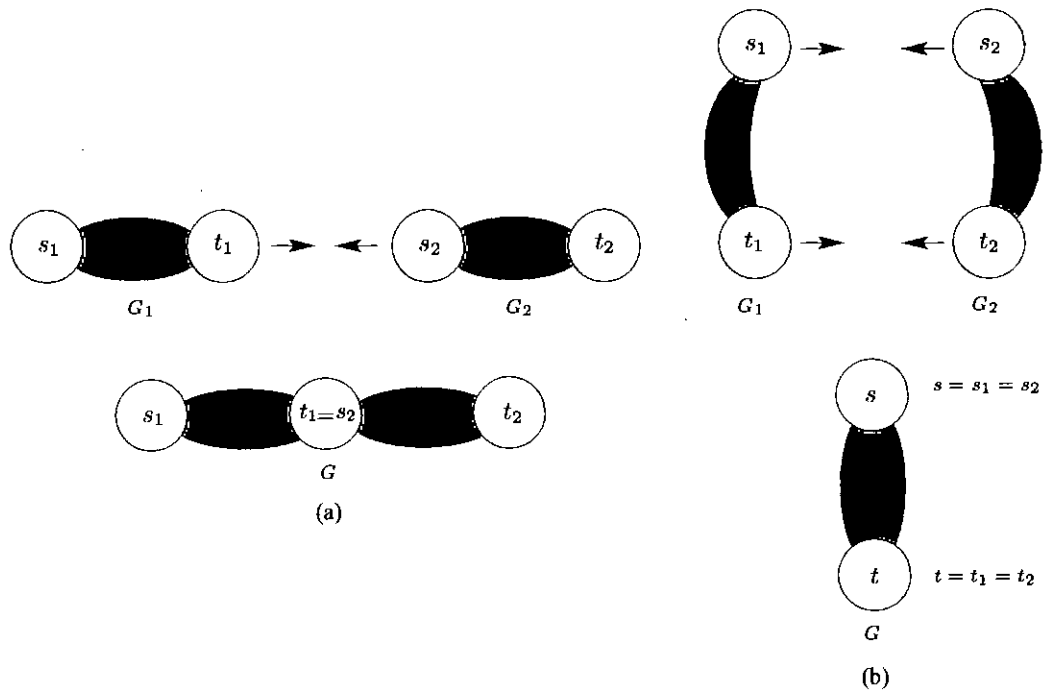


Figure 2.8: A series-parallel graph G composed from G_1 and G_2 (a) with series connection, and (b) with parallel connection.

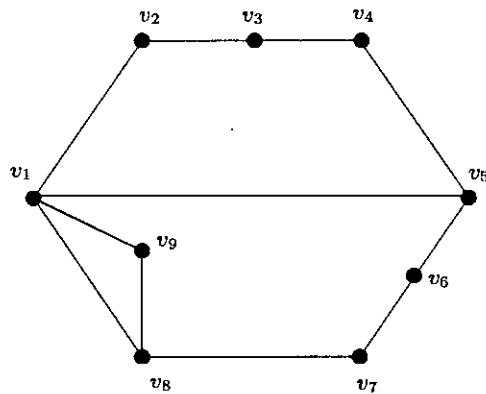


Figure 2.9: A series-parallel graph G .

labeled s and p are called s -nodes and p -nodes, respectively.

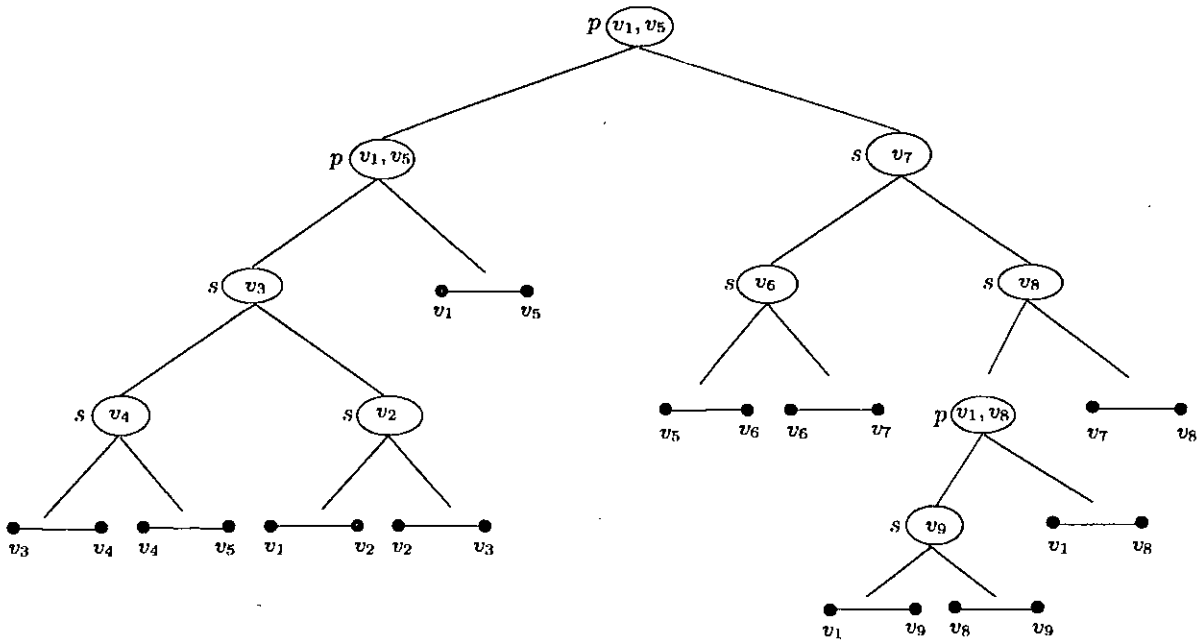


Figure 2.10: Binary decomposition tree T_b of a series-parallel graph G in Figure 2.9.

2.3 Complexity of Algorithms

The efficiency or complexity of an algorithm is determined by the amount of resources (such as time and storage) necessary to execute it. Generally, it is defined as a function relating the input length n to the number of steps (time complexity) or storage locations (space or memory complexity) required to execute the algorithm. In theoretical analysis of algorithms it is common to estimate their complexity in asymptotic sense, i.e., to estimate the complexity function for reasonably large length of input n . For example, since binary search is said to run an amount of steps proportional to a logarithm, its complexity of the running time is defined by $O(\log(n))$. If the running time of an algorithm is bounded by $O(n)$, it is said to be

a *linear-time* algorithm.

2.3.1 Complexity Classes: \mathcal{P} and \mathcal{NP}

A problem is said to have a *polynomial-time* algorithm 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 superpolynomial time are intractable or hard. Based upon the running time of algorithms, next we define complexity classes. The class \mathcal{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; the class \mathcal{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 \mathcal{P} is also in \mathcal{NP} , since if a problem is in \mathcal{P} then we can verify it in polynomial time.

2.3.2 \mathcal{NP} -Complete Problem

Here, we are mainly interested in another class of problems, called *\mathcal{NP} -complete problems* (or \mathcal{NPC}), which can be loosely described as the hardest problems in \mathcal{NP} and therefore they are the least likely to be in \mathcal{P} . No polynomial-time algorithm has yet been discovered for an \mathcal{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 \mathcal{NP} -complete if it is complete for \mathcal{NP} , meaning that:

- (1) it is in \mathcal{NP} , and
- (2) it is \mathcal{NP} -hard, i.e. every other problem in \mathcal{NP} is polynomial-time reducible

to it.

“Polynomial-time reducible” here 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 \mathcal{NP} problem A is in fact an \mathcal{NP} -complete problem it is sufficient to show that an already known \mathcal{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.

2.4 Approximation Algorithm and Approximation Ratio

At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size. 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*.

Depending on the problem, maximization or minimization, an optimal solution may be defined as one with maximum possible cost or one with minimum possible cost. An approximation ratio is a measure of goodness of the approximation solution with the optimal solution of the problem. An algorithm for a problem has an *approximation ratio* of $\rho(n)$ if, for any input size n , the cost C of the solution produced by the algorithm is within factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \quad (2.1)$$

An algorithm that achieves an approximation ratio of $\rho(n)$ is called $\rho(n)$ -approximation algorithm. For a maximization problem, $0 < C \leq C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution. Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of an approximate solution is larger than the cost of the optimal solution. Since all solutions are assumed to have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* < 1$ implies $C^*/C > 1$. Therefore, a 1-approximation algorithm produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

Chapter 3

2-Vertex-Separator Tree

Since a series-parallel graph is a partial 2-tree it is known that each series-parallel graph has a 3-vertex-separator tree[14]. In this chapter we construct a 2-vertex-separator-tree of a series-parallel graph using its special structure. In Section 3.1, we show that a series-parallel graph has a 2-vertex-separator tree. A series-parallel graph can be disconnected by removing at most two vertices. However, disconnected components that do not have the series-parallel structure may be yielded by this process. So we show that every such component has at least one cut-vertex. This immediately proves that a series-parallel graph has a 2-vertex-separator tree. Next in Section 3.2, we describe an algorithm for constructing a 2-vertex-separator tree of a simple series-parallel graph using binary decomposition tree of the series-parallel graph. We also analyze the time-complexity of the algorithm.

3.1 Preliminaries

A single edge graph is a series-parallel graph. Larger series-parallel graphs can be composed from smaller series-parallel graphs either using series connection or parallel

connection. If a series-parallel graph G with terminals s and t was composed with series connection from two smaller series-parallel graphs G_1 with terminals s_1 and t_1 and G_2 with terminals s_2 and t_2 (see Figure 2.8(a)), G can be decomposed into components by removing the single vertex ($s_2 = t_1$) through which G was composed. Again, if a series-parallel graph G with terminals s and t was composed with parallel connection from two smaller series-parallel graphs G_1 with terminals s_1 and t_1 and G_2 with terminals s_2 and t_2 (see Figure 2.8(b)), G can be decomposed into components by removing the two vertices ($s = s_1 = s_2$ and $t = t_1 = t_2$) through which G was composed. Figure 3.1 and Figure 3.2 illustrate series and parallel decomposition of a series-parallel graph. We then have the following lemma directly from the definition of a series-parallel graph.

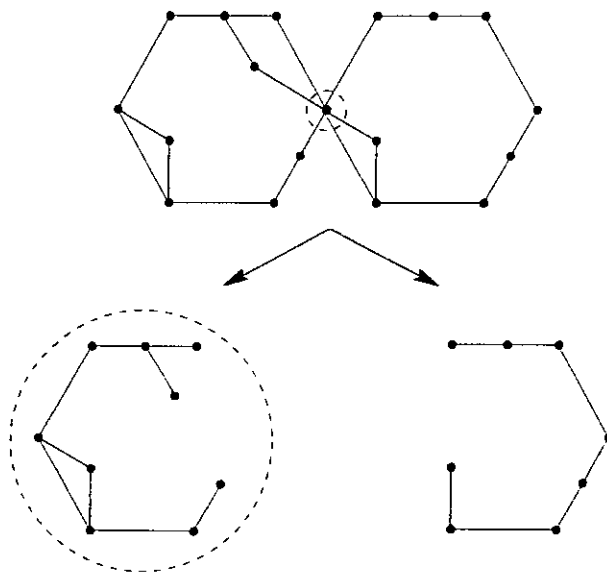


Figure 3.1: A series-parallel graph G is decomposed by removing a vertex.

Lemma 3.1.1 *A series-parallel graph can be decomposed into components either by removing a single vertex if the graph was composed with series connection or by removing two vertices if the graph was composed with parallel connection. \square*

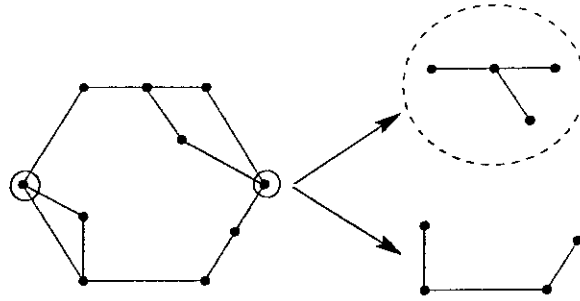


Figure 3.2: A series-parallel graph G is decomposed by removing two vertices.

After decomposing the series-parallel graph according to Lemma 3.1.1, the resulting components may lose series-parallel structure. Figures 3.1 and 3.2 illustrate that due to decomposition of series-parallel graph according to Lemma 3.1.1 the resulting components may or may not have series-parallel structure. In Figures 3.1 and 3.2 the circled components do not have the series-parallel structure. After decomposition of series-parallel graph if the resulting component loses its series-parallel structure, then there must be an inner parallel connection of the original series-parallel graph with one terminal removed. Then the other terminal which is not still removed in the resulting component becomes the cut vertex of that component. We next have the following lemma:

Lemma 3.1.2 *Let G be a series-parallel graph. If a resulting component D does not have series-parallel structure as a result of decomposition by removing one vertex (series connection) or two vertices (parallel connection) from G , then D must have a cut vertex.*

Proof. Let G be a series-parallel graph composed from two series-parallel graphs G_1 and G_2 .

Let G be composed from G_1 and G_2 through the series vertex u . Let D be a component that does not have series-parallel structure as a result of removing u

from G . Without loss of generality assume that D is a subgraph of G_2 . Let the graph G_2 be composed from either series connection through the vertex v from two smaller series-parallel graphs G_3 and G_4 , or only G_3 , where G_3 is a series-parallel graph composed from parallel connection with the terminal vertices u and v . Then G_3 can be disconnected by removing u and v . Since the component D is obtained by removing u from G , G_2 is a subgraph of G and D is a subgraph of G_2 , D contains the vertex v but not u . So if v is removed from D , D will be disconnected. Thus v is a cut vertex in D . (See Figures 3.3 and 3.4.)

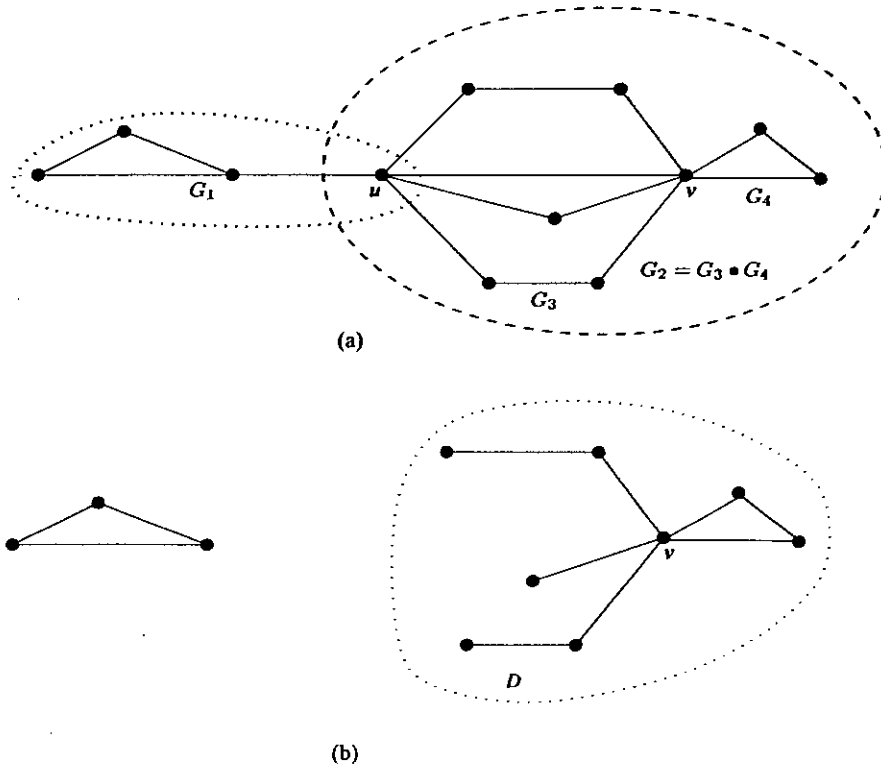


Figure 3.3: (a) A series-parallel graph G composed from G_1 and G_2 (composed from G_3 and G_4) through u , and (b) resulting components after removing u .

Let G be composed from G_1 and G_2 through the parallel vertices u and v . Let D be a component that does not have series-parallel structure as a result of removing u

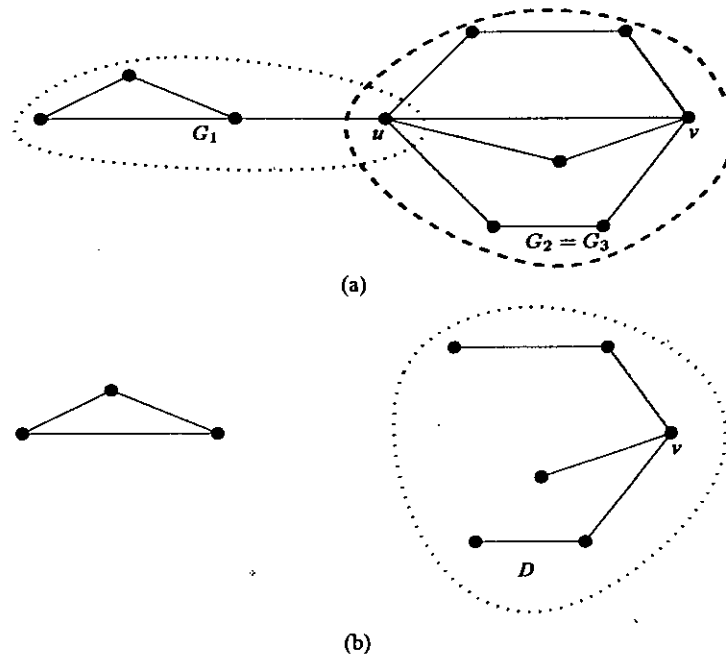


Figure 3.4: (a) A series-parallel graph G composed from G_1 and G_2 (composed from only G_3) through u , and (b) resulting components after removing u .

and v from G . Without loss of generality assume that D is a subgraph of G_2 . Let the graph G_2 be composed by series connection through the vertex w from two smaller series-parallel graphs G_3 and G_4 , where G_3 is a series-parallel graph composed from parallel connection with the terminal vertices u and w . Then G_3 can be disconnected by removing u and w . Since the component D is obtained by removing u from G , G_2 is a subgraph of G and D is a subgraph of G_2 , D contains the vertex w but not u . So if w is removed from D , D will be disconnected. Thus w is a cut vertex in D . (See Figure 3.5.)

So after decomposition of a series-parallel graph G when a resulting component D loses its series-parallel structure, there must be an inner parallel connection of G with one terminal removed. Then the other terminal which is not still removed in D becomes the cut vertex of the component. □

After removing all cut vertices from a component which is not a series-parallel graph may result in a series-parallel graph or a graph (if the resulting graph is not a series-parallel graph) with at least one cut vertex.

Now from Lemmas 3.1.1 and 3.1.2 we have the following theorem.

Theorem 3.1.3 *A series-parallel graph has a 2-vertex-separator tree.* □

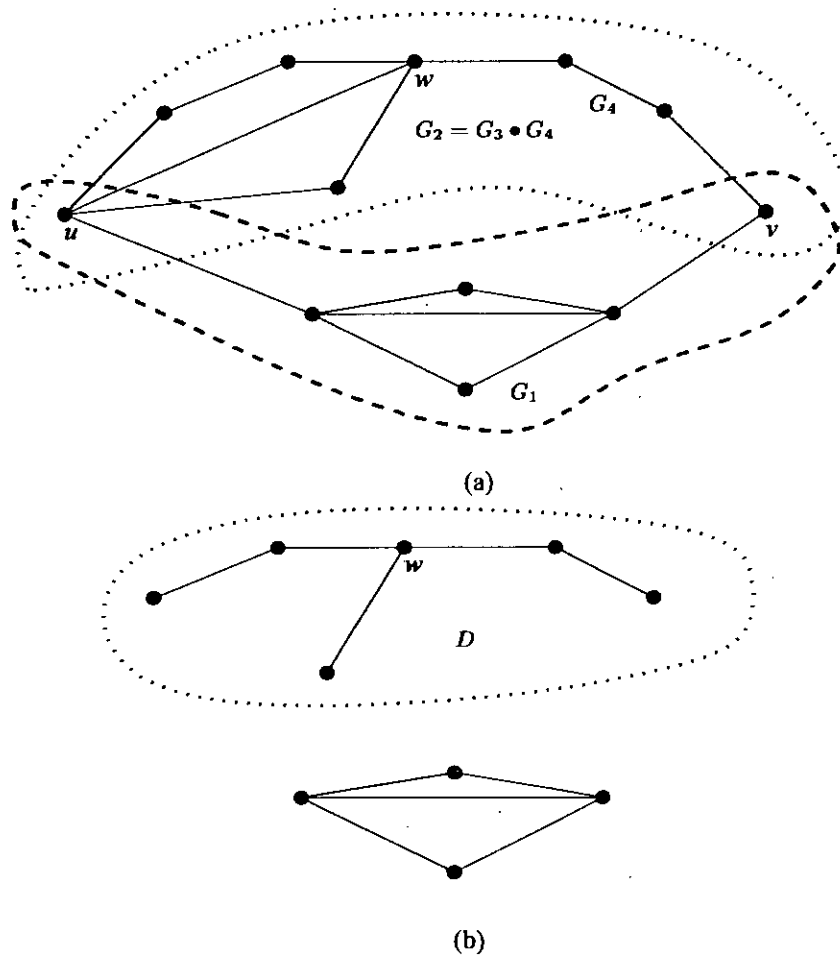


Figure 3.5: (a) A series-parallel graph G composed from G_1 and G_2 through u and v , and (b) the resulting components after removing u and v .

3.2 The Algorithm

Now we have the algorithm `SP_2_VertexSeparatorTree` to construct a 2-vertex-separator tree of a series-parallel graph. Let $T_b = (V_{T_b}, E_{T_b})$ be a binary decomposition tree of a series-parallel graph G . (See Figure 3.6(c)). Our algorithm constructs 2-vertex-separator tree T (see Figure 3.6(b)) of G using T_b . Let $T_b(x)$ be the subtree of T_b rooted at node x . Every leaf x of T_b represents a subgraph of G induced by two vertices s and t connected by the edge (s, t) and let $S_x = \{s, t\}$ be the set of terminals of G_x . We associate a subgraph $G_x = (V_x, E_x)$ of G with each node x of T_b , where

$$V_x = \cup \{S_y \mid y = x \text{ or } y \text{ is a descendent of } x \text{ in } T_b\}$$

$$E_x = \{e_y \mid y \text{ is a leaf node in } T_b(x)\}$$

The graph associated with the root of T_b is the given graph G itself. The left child and right child of an internal node x in T_b are denoted by y and z , respectively. Every internal node x in T_b is either a s -node or a p -node and contains one or two vertices of G to disconnect the graphs associated with node y and node z , that is G_y and G_z , respectively.

Again let A be an array with n entries. Each entry at index i of A corresponds to vertex v_i of G . $A[i]$ can be defined as follows:

$$A[i] = \begin{cases} 0 & \text{if the vertex } v_i \text{ is considered for 2-vertex-separator tree, and} \\ 1 & \text{otherwise.} \end{cases}$$

Initially each entry of A is initialized with 1. `SP_2_VertexSeparatorTree` is a recursive algorithm that traverses a single node in every run of it. The algorithm traverses the nodes of T_b in preorder fashion. Every node x in T_b contains at most two vertices, and these vertices may exist or may not exist if it is already considered for the 2-vertex-separator tree and the algorithm checks it from A .

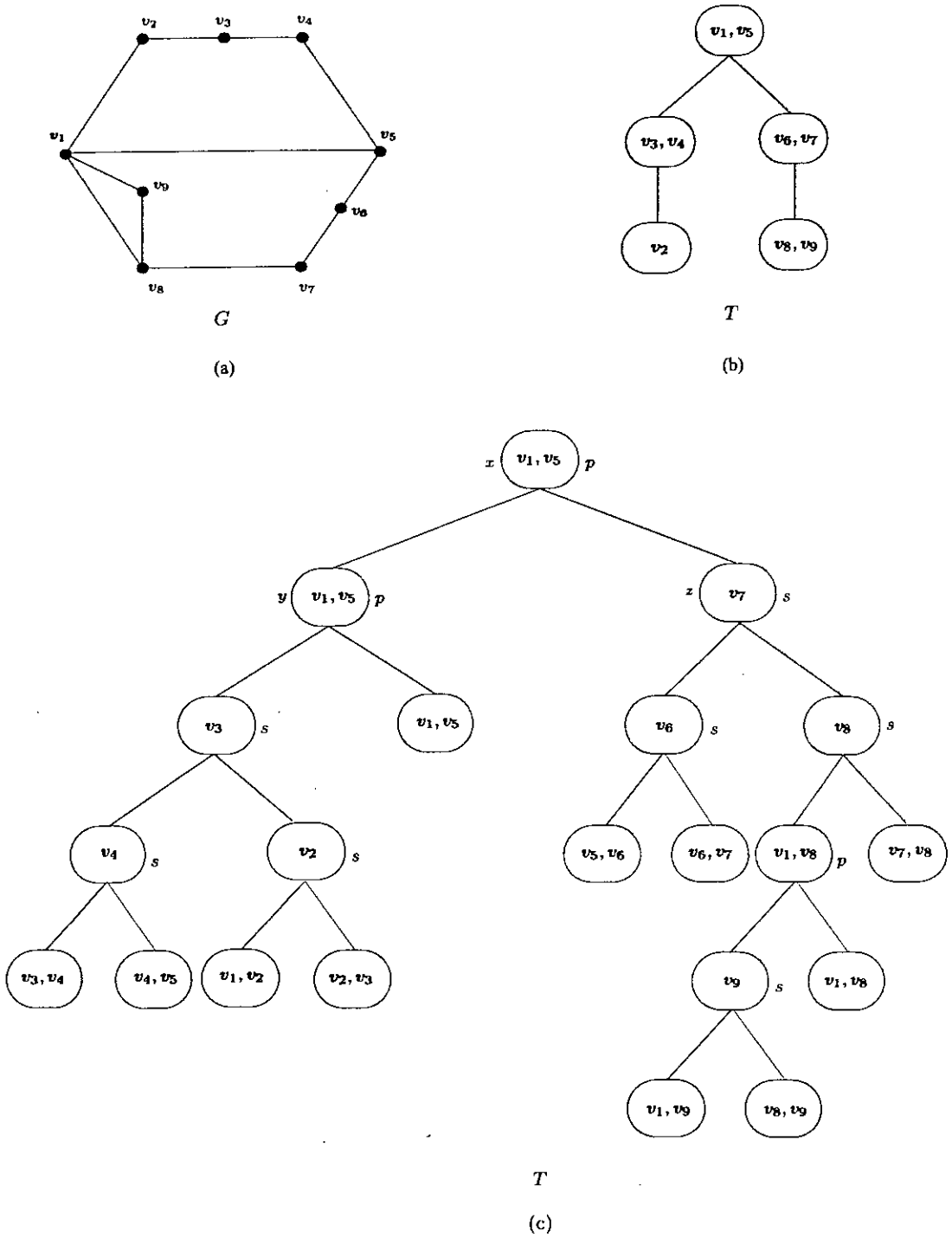


Figure 3.6: (a) A series-parallel graph G , (b) a 2-vertex-separator tree T of G , and (c) a binary decomposition tree T_b of G .

`SP_2_VertexSeparatorTree` takes a node x as input, where x is the current traversing node in T_b . `SP_2_VertexSeparatorTree` takes another input i , where i is either the index of the vertex $v_i, 1 \leq i \leq n$, which is not yet associated with any node of the 2-vertex-separator tree but is already been considered while executing for its ancestor node or -1 . `SP_2_VertexSeparatorTree` constructs 2-vertex-separator tree T of G , where every internal node of T associates two vertices of G . So when there is only one vertex in any run of the algorithm the vertex is not associated with any node of T and the index of the vertex is transmitted to its descendent for future association to any node.

The algorithm `SP_2_VertexSeparatorTree` works in two phases. In first phase it constructs T with a single or two nodes using the algorithms `From_Leafnode` (given later), `From_Seriesnode` (given later), `From_Parallelnode`(given later). In the second phase it recursively calls `SP_2_VertexSeparatorTree` for its child nodes and modifies T using 2-vertex-separator tree that it gets from the recursive return of `SP_2_VertexSeparatorTree`. To modify, it uses `Adjust_2vertexSeparatorTree` (given later) and `Re_Adjust_2vertexSeparatorTree` (given later). Now to construct the 2-vertex-separator tree of a series-parallel graph G we call `SP_2_VertexSeparatorTree` with x and i , where x is the root node in T_b and i is -1 .

Algorithm *SP_2_VertexSeparatorTree*(x, i)

Input: A node x in binary decomposition tree T_b , and an index variable i .

Output: Return a 2-vertex-separator tree T of G .

begin

- 1 $flag := 0;$
- 2 $two_node := 0$
- 3 **if** x is a leaf node **then**
- 4 $From_Leafnode(i);$

```

5   return 2-vertex-separator tree  $T$ ;
6   else if  $x$  is a  $s$ -node then
7     From_Seriesnode( $i$ );
8   else { $x$  is a  $p$ -node}
9     From_Parallelnode( $i$ );
10  if node  $x$  in  $T_b$  has a left child  $y$  then
11     $T' \leftarrow \text{SP\_2vertexSeparatorTree}(y, i)$ ;
12    if two_node == 1 then
13      if  $T'$  is a tree with a node  $r$  and  $r$  is associated with no vertex then
14        two_node := 0;
15      else { $T'$  is a tree with a node  $r$  and  $r$  is associated with one or two
16        vertices}
17        modify  $T$  by making the node  $r$  in  $T$  as the parent of the root node of
18         $T'$ ;
19      else { $T'$  is a tree with a node  $r$  and  $r$  is associated with one or two vertices}
20       $T \leftarrow \text{Adjust\_2vertexSeparatorTree}(T, T', i, 0)$ ;
21  if node  $x$  in  $T_b$  has a right child  $z$  then
22     $T'' \leftarrow \text{SP\_2vertexSeparatorTree}(z, i)$ ;
23    if  $T''$  is a tree with a node  $r$  and  $r$  is associated with one or two vertices
24      then
25         $T \leftarrow \text{Adjust\_2vertexSeparatorTree}(T, T'', i, \textit{two\_node})$ ;
26  if ( $y$  exists and  $T'$  is a tree with a node  $r$  and  $r$  is associated with no
27    vertex) or ( $z$  exists and  $T''$  is a tree with a node  $r$  and  $r$  is associated
28    with no vertex) and  $i \neq -1$  then
29    if  $x$  is a  $s$ -node then
30      if  $i = i'$  then
31        flag := 1;

```

```

    else { $x$  is a  $p$ -node}
27   if  $i = i'$  or  $i = i''$  then
28      $flag := 1$ ;
29   if  $flag = 1$  then
30      $T \leftarrow \text{Re\_Adjust\_2vertexSeparatorTree}(T, i)$ ;
31   return 2 vertex-separator tree  $T$ ;
end

```

The algorithm `SP_2_VertexSeparatorTree` based on the Theorem 3.1.3 correctly constructs a 2-vertex-separator tree of G since there is no more than 2 vertices associated with each node of T . If x is a leaf node of T_b then `From_Leafnode` constructs T with a single or two nodes. If there is no transmitted node, that is i equals to -1 , then it constructs T with a single node associated with two vertices (if both vertices in x exist) or one vertex (if one vertex in x exists). But when i is the index of the vertex $v_i, 1 \leq i \leq n$, which is not yet associated with any node of the 2-vertex-separator tree but is already been considered while executing for its ancestor node then it constructs T with two nodes (if both vertices in x exist): one associated with two vertices and the other with single vertex, and with single node associated with two vertices (if one vertex in x exists). Otherwise, T is a tree with a node associated with no vertex.

Algorithm *From_Leafnode*(i)

Input: An index variable i .

Output: Return a 2-vertex-separator tree T , and an index i .

begin

- 1 make a node r and associate no vertex with it;
 - let $v_{i'}$ and $v_{i''}$ be the endpoints of the edge in G corresponding to the leaf node of T_b ;

```

2  if  $A[i'] = 1$  and  $A[i''] = 1$  then
3       $A[i'] := 0$ ;  $A[i''] := 0$ ;
4      if  $i = -1$  then
5          associate vertices  $v_{i'}$  and  $v_{i''}$  to the node  $r$ ;
          else  $\{i$  is the index of a vertex  $v_i$  in  $\{v_1, v_2, \dots, v_n\}\}$ 
6              associate vertices  $v_i$  and  $v_{i'}$  to the node  $r$ ;
7              make another node  $q$  and associate the vertex  $v_{i''}$  with it;
8      else if  $A[i'] = 1$  or  $A[i''] = 1$  then
          without loss of generality assume that  $A[i'] = 1$   $\{\text{All parameters are similarly}$ 
          handled for  $A[i''] = 1\}$ 
9           $A[i'] := 0$ ;
10         if  $i = -1$  then
11             associate the vertices  $v_{i'}$  to the node  $r$ ;
             else  $\{i$  is the index of a vertex  $v_i$  in  $\{v_1, v_2, \dots, v_n\}\}$ 
12                 associate vertices  $v_i$  and  $v_{i'}$  to the node  $r$ ;
13         if  $A[i'] = 1$  and  $A[i''] = 1$  and  $i \neq -1$  then
14             let  $T$  be the tree with the nodes  $r$  and  $q$ , where  $r$  is the parent node of  $q$ ;
          else
15             let  $T$  be the tree with the node  $r$  only;
16         return 2-vertex-separator tree  $T$ , and the index  $i$ ;
end

```

If x is a s -node of T_b then `From_Seriesnode` constructs T with a single node. If there is no transmitted node, that is i equals to -1 , then it constructs T with a single node associated with one vertex (if the vertex in x exists). But when i is the index of the vertex v_i , $1 \leq i \leq n$, which is not yet associated with any node of the 2-vertex-separator tree but is already been considered while executing for its ancestor node then it constructs T with with single node associated with two

vertices (if the vertex in x exists). Otherwise, T is a tree with a node associated with no vertex. `From_Seriesnode` also determines the value of i for next recursive call of `SP_2_VertexSeparatorTree`.

Algorithm *From_Seriesnode*(i)

Input: An index variable i .

Output: Return a 2-vertex-separator tree T , and an index i .

begin

```

1  make a node  $r$  and associate no vertex with it;
   let  $v_{i'}$  be the vertex in  $G$  through which series connection was made;
2  if  $A[i'] = 1$  then
3     $A[i'] := 0$ ;
4    if  $i = -1$  then
5       $i := i'$ ;
   else { $i$  is the index of a vertex  $v_i$  in  $\{v_1, v_2, \dots, v_n\}$ }
6     $i = -1$ ;
7    associate vertices  $v_i$  and  $v_{i'}$  to the node  $r$ ;
8  let  $T$  be the tree with the node  $r$  only;
9  return 2-vertex-separator tree  $T$ , and the index  $i$ ;
end

```

If x is a p -node of T_b then `From_Parallelnode` constructs T with a single node. If there is no transmitted node, that is i equals to -1 , then it constructs T with a single node associated with two vertices (if both vertices in x exist). But when i is the index of the vertex v_i , $1 \leq i \leq n$, which is not yet associated with any node of the 2-vertex-separator tree but is already been considered while executing for its ancestor node then it constructs T with single node associated with two vertices (if one vertex or two vertices in x exists). Otherwise, T is a tree with a

node associated with no vertex. *From_Parallelnode* also determines the value of i for the next recursive call of *SP_2_VertexSeparatorTree* and also set decision variable *two_node* when i not equals to -1 and both vertices in x exist. By setting *two_node* to 1 it implies that there is a probability of two nodes of T for this x in T_b . If there is a vertex in any node of $T(y)$ that is not yet considered, then *two_node* will remain 1. The variable *two_node* is used in the modification phase.

Algorithm *From_Parallelnode*(i)

Input: An index variable i .

Output: Return 2-vertex-separator tree T , an index i , and a variable *two_node*.

begin

- 1 make a node τ and associate no vertex with it;
let $v_{i'}$ and $v_{i''}$ be the vertices in G through which parallel connection was made;
- 2 **if** $A[i'] = 1$ and $A[i''] = 1$ **then**
- 3 $A[i'] := 0$; $A[i''] := 0$;
- 4 **if** $i = -1$ **then**
- 5 associate vertices $v_{i'}$ and $v_{i''}$ to the node τ ;
- 6 **else** $\{i$ is the index of a vertex v_i in $\{v_1, v_2, \dots, v_n\}\}$
- 7 $i := i''$;
- 8 $two_node := 1$;
- 9 associate vertices v_i and $v_{i'}$ to the node τ ;
- 9 **else if** $A[i'] = 1$ or $A[i''] = 1$ **then**
without loss of generality assume that $A[i'] = 1$ {All parameters are similarly handled for $A[i''] = 1$ }
- 10 $A[i'] := 0$;
- 11 **if** $i = -1$ **then**

```

12      $i := i'$ ;
        else  $\{i$  is the index of a vertex  $v_i$  in  $\{v_1, v_2, \dots, v_n\}\}$ 
13      $i := -1$ ;
14     associate vertices  $v_i$  and  $v_{i'}$  to the node  $r$ ;
15     let  $T$  be the tree with the node  $r$  only;
16     return 2-vertex-separator tree  $T$ , the index  $i$ , and the variable  $two\_node$ ;
    end

```

Now `Adjust_2_VertexSeparatorTree` and `Re_Adjust_2_VertexSeparatorTree` modify T that by merging T with the 2-vertex-separator tree that it gets from recursive return of `SP_2_VertexSeparatorTree` for the child nodes of x in T . `Adjust_2_VertexSeparatorTree` also determines the value of i for the next recursive call of `SP_2_VertexSeparatorTree`.

Algorithm *Adjust_2_VertexSeparatorTree*(T, T_child, i, two_node)

Input: A 2-vertex-separator tree T , another 2-vertex-separator tree T_child , an index variable i , and another variable two_node .

Output: Return a 2-vertex-separator tree T and an index i .

```

begin
1  if  $two\_node = 0$  then
2    if  $i \neq -1$  then
3       $T := T\_child$ ;
4       $i := -1$ ;
5    else  $\{i=-1\}$ 
6    if the root node  $r'$  of  $T'$  is associated with no vertex then
7      if  $r'$  has children  $\{a_1, a_2, \dots, a_k\}$  then
8        remove  $r'$  and make  $\{a_1, a_2, \dots, a_k\}$  the children of the node  $r$  in  $T$ ;
9    else

```

```

10     modify  $T$  by making the root  $r'$  of  $T'$  as the child of node  $r$  in  $T$ ;
11  else {two_node = 1}
12    if the root node  $r'$  of  $T'$  is associated with no vertex then
        let  $q$  be the child of node  $r$  in  $T$ ;
13    if  $r'$  has children  $\{a_1, a_2, \dots, a_k\}$  then
14      remove  $r'$  and make  $\{a_1, a_2, \dots, a_k\}$  the children of the node  $q$  in  $T$ ;
15    else
16      modify  $T$  by making the root  $r'$  of  $T'$  as the child of node  $q$  in  $T$ ;
17  return 2-vertex-separator tree  $T$ , and the index  $i$ ;
    end

```

Re_Adjust_2_VertexSeparatorTree is used to adjust T in a special case when all the 2-vertex-separator tree that are returned contain a node associated with no vertex, the present value of i is not equal to -1 and vertex v_i exists in x . In that case Re_Adjust_2_VertexSeparatorTree add a new node associated with v_i in T .

Algorithm *Re_Adjust_2_VertexSeparatorTree*(T, i)

Input: A 2-vertex-separator tree T , and an index variable i .

Output: Return a 2-vertex-separator tree T

```

begin
1  if  $T$  is a tree with a node  $r$  and  $r$  is associated with no vertex then
2    associate the vertex  $v_i$  to node  $r$  of  $T$ ;
3  else
4    make a new node  $q$  and associate the vertex  $v_i$  with it;
5    modify  $T$  by making  $q$  as a child of  $r$ ;
6  return 2-vertex-separator tree  $T$ ;
end

```

A series-parallel graph can be represented by a binary decomposition tree in

linear time [25]. Every operations including conditional statements of the algorithms `From_Leafnode`, `From_Seriesnode`, `From_Parallelnode`, `Adjust_2vertexSeparatorTree` and `Re_Adjust_2vertexSeparatorTree` execute in constant time. Now to construct a 2-vertex-separator tree of G , `SP_2_VertexSeparatorTree` traverses the binary decomposition tree T_b in preorder fashion. Since the number of nodes in the binary decomposition tree is $O(n)$, the complexity of `SP_2_VertexSeparatorTree` is $O(n)$, where n is the number of vertices in G .

3.3 Conclusion

In this chapter we present a linear-time algorithm for constructing a 2-vertex-separator tree of a simple series-parallel graph using its special structure. The height of the 2-vertex-separator tree is $\leq n/2$. We shall use this 2-vertex-separator tree in the next chapter for obtaining approximation algorithm to find an edge-ranking of a series-parallel graph. Obtaining a 2-vertex-separator tree immediately improves the upper bound of the optimal vertex-ranking number and thereby the running time of the known best algorithm that finds the optimal vertex-ranking of a series-parallel graph.

Chapter 4

Approximation Algorithm

This chapter deals with the approximation algorithm for finding the edge-ranking of a series-parallel graph. The algorithm is based on the 2-vertex-separator tree discussed in Chapter 3. This chapter is organized as follows. In Section 4.1, first we define some terms related to the algorithm and then propose the algorithm. We also analyze the time-complexity and give the correctness of the algorithm. Finally, we illustrate our algorithm step by step using an example. In Section 4.2, we calculate the approximation ratio. To do that we first find the lower bound of the optimal edge-ranking number of series-parallel graphs and then the upper bound of edge-ranking number of series-parallel graphs used by our approximation algorithm. We also discuss the reason behind the deviation from optimality of our approximation algorithm.

4.1 The Algorithm

Let T be a 2-vertex-separator tree of a series-parallel graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$ and $E = \{e_1, e_2, \dots, e_m\}$. Let X_x be the set of vertices in node

x of T . In Figure 4.1, $X_x = \{v_1, v_5\}$, $X_{y_1} = \{v_3, v_4\}$, and $X_{y_2} = \{v_6, v_7\}$. We associate a subgraph $G_x = (V_x, E_x)$ of G with each node x of T , where

$$V_x = \bigcup\{u \mid u \in X_y \text{ and } y = x \text{ or } y \text{ is a descendent of } x \text{ in } T\}$$

$$E_x = \{(u, v) \mid u, v \in X_y \text{ and } y = x \text{ or } y \text{ is a descendent of } x \text{ in } T\}$$

The graph associated with node x is $G_x = (V_x, E_x)$ as shown in Figure 4.1(b). The children of node x in T is labeled with y_i , $1 \leq i \leq d$, if x has d children. Here node x has two child nodes: y_1 and y_2 . Graphs associated with y_1 and y_2 are $G_{y_1} = (V_{y_1}, E_{y_1})$ and $G_{y_2} = (V_{y_2}, E_{y_2})$ as shown in Figure 4.1(c) and Figure 4.1(d), respectively.

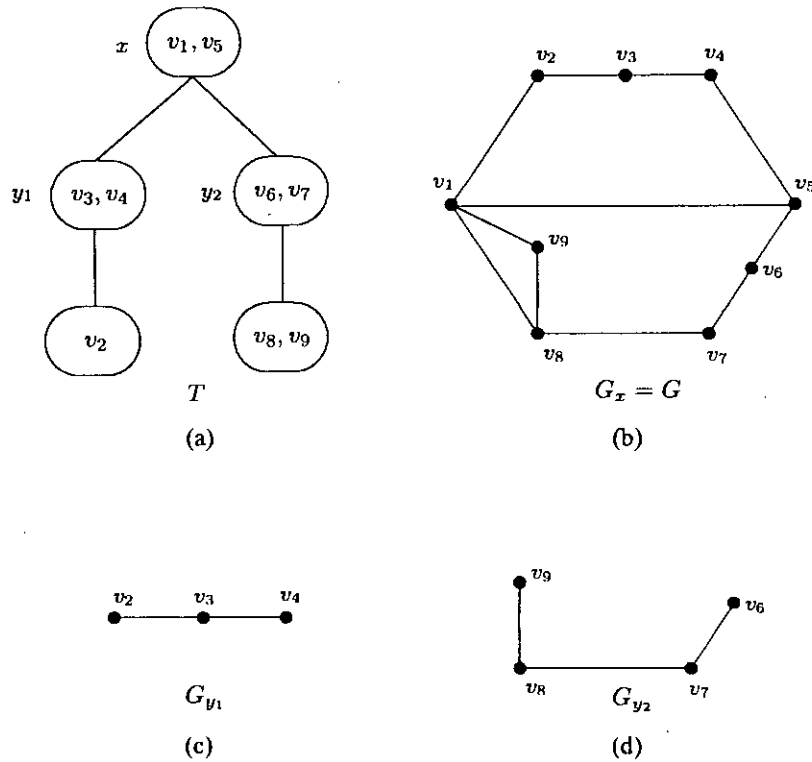


Figure 4.1: (a) A 2-vertex-separator tree T of graph G , (b) graph G_x associated with node x of T , (c) subgraph G_{y_1} of G associated with node y_1 of T , and (d) subgraph G_{y_2} of G associated with node y_2 of T .

Let us define the set F_x as follows: the set of edges connecting vertices in X_x to vertices in V_{y_i} , $1 \leq i \leq d$ and also edges between vertices in X_x . In Figure 4.1, we have $F_x = \{(v_1, v_2), (v_1, v_8), (v_1, v_9), (v_5, v_4), (v_5, v_6), (v_1, v_6)\}$. Let r_i , $1 \leq i \leq d$ be the largest rank used for ranking the edges in E_{y_i} . We also define r as follows:

$$r = \max\{r_i | y_i \text{ is a child of } x \text{ and } 1 \leq i \leq d\}.$$

To rank the edges of G we call `SP_Approx_Rank` with G_r , where r is the root of T and G_r is the subgraph associated with r . Note that is G_r is actually the given series-parallel graph G .

Algorithm *SP_Approx_Rank*(G_x)

Input: A graph $G_x = (V_x, E_x)$, the subgraph of G corresponding to node x of T .

Output: An edge-ranking of G_x .

begin

1 **if** x is a leaf node and $|E_x| = 1$ **then**

2 rank the edge in E_x with rank 1;

else

3 **for** each child node y_i , $1 \leq i \leq d$, of x **do**

4 `SP_Approx_Rank`(G_{y_i});

 Let F_x be the set of edges connecting vertices in X_x to vertices in V_{y_i} ,

$1 \leq i \leq d$ and also edges between vertices in X_x and r be defined as

$r = \max\{r_i | y_i \text{ is a child of } x \text{ and } 1 \leq i \leq d\}$;

5 rank sequentially the edges in F_x with different ranks starting from rank

$r + 1$;

end

For a leaf node of T the algorithm ranks the edge in its associated subgraph in $O(1)$ time. Since for every internal node x of T there can be at most 2 vertices in

$X_x \subseteq V_x$, the number of edges to be ranked is $d(v) + d(w)$, where $v, w \in X_x$. So for each internal node of T the algorithm takes $O(d(v))$ time to rank these edges in Step 5. So the overall running time of the algorithm is $O(\sum_{v \in V} d(v)) = O(|E|) = O(n)$.

Lemma 4.1.1 *SP_Approx_Rank finds an edge-ranking of a series-parallel graph G_x correctly.*

Proof. Let T be a 2-vertex-separator tree of a series-parallel graph G . G_x is a graph associated with node x of T . The algorithm labels the edges (if exists) contained in the subgraph of a leaf node of T with 1 as there can be maximum one edge in the subgraph of a leaf node. So if x is a leaf node of T then SP_Approx_Rank(G_x) finds an edge-ranking of a series-parallel graph G_x correctly. The edges in F_x are ranked with labels greater than the labels used in ranking the edges of E_{y_i} , where $y_i, 1 \leq i \leq d$ is a child of x . Let $\{G_{y_1} = (V_{y_1}, E_{y_1})\}, \{G_{y_2} = (V_{y_2}, E_{y_2})\}, \dots, \{G_{y_d} = (V_{y_d}, E_{y_d})\}$ are the d subgraphs associated with nodes y_1, y_2, \dots, y_d , where y_1, y_2, \dots, y_d are the child nodes of x . Here according to the algorithm, it is possible that some of the edges from $\{e_1, e_2, \dots, e_d\}$, where $e_1 \in E_{y_1}, e_2 \in E_{y_2}, \dots, e_d \in E_{y_d}$, have the same label. But all paths between this two edges contain one of the edges from the edges in F_x as the subgraphs $G_{y_1}, G_{y_2}, \dots, G_{y_d}$ are connected through the edges in F_x . The labels of all edges in F_x are greater than the label of all edges in $E_{y_1}, E_{y_2}, \dots, E_{y_d}$. So if x is an internal node of T then SP_Approx_Rank also finds an edge-ranking of a series-parallel graph G_x correctly. \square

If we call SP_Approx_Rank with G_r the algorithm traverses the tree in postorder and while traversing rank the edges of the subgraph associated with each node. Finally the algorithm returns the edge-ranking of $G_r = G$ when the traversal of the T is finished.

4.1.1 An Example

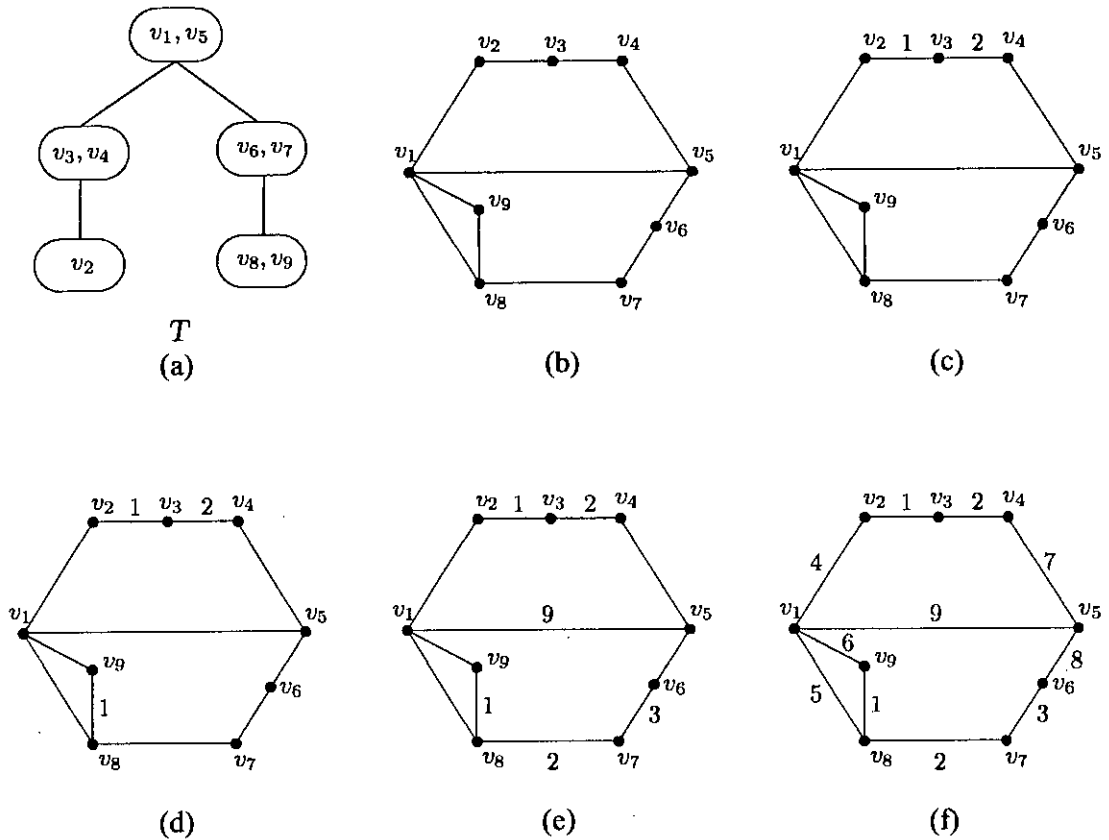


Figure 4.2: (a) A 2-vertex-separator tree of G , (b) A series-parallel graph G and (c)-(f) after each call of `SP_Approx_Rank`.

Now we illustrate the approximation algorithm for edge-ranking of a series-parallel graph with an example. A series-parallel graph G is given in Figure 4.2(b) and the 2-vertex-separator tree T for the graph is shown in Figure 4.2(a). We rank the edges of G using `SP_Approx_Rank`. The algorithm `SP_Approx_Rank` traverses T in postorder fashion. The algorithm starts from the root node and goes to the leaf node that contains the vertex v_2 . Since there is no edge in the graph associated with this node, next it goes to the node that contains vertices v_3 and v_4 . Now

$F_x = \{(v_2, v_3), (v_3, v_4)\}$. Since in this stage $r = 0$, the algorithm ranks (v_2, v_3) with $r + 1$, that is 1 and (v_3, v_4) with $r + 2$, that is 2, as shown in Figure 4.2(c). The traverse of left subtree of the root node of T is now complete. Then the ranking of the graph associated with right child of the root node should be completed. The subgraph associated with the leaf node contains the only edge (v_8, v_9) and it is ranked with 1(Figure 4.2(d)). Now the algorithm visits the node that contains the vertices (v_6, v_7) . Here $F_x = \{(v_7, v_8), (v_6, v_7)\}$. Edges (v_7, v_8) and (v_6, v_7) are ranked with 2 and 3, respectively(Figure 4.2(e)), since the rank is already used for (v_8, v_9) is 1. For the root node $F_x = \{(v_1, v_2), (v_1, v_8), (v_1, v_9), (v_4, v_5), (v_5, v_6), (v_1, v_5)\}$. The maximum rank, r used in the graph associated with the child nodes of the root node is 3. So to rank the edges in F_x the algorithm uses different ranks starting from 4. The edges $(v_1, v_2), (v_1, v_8), (v_1, v_9), (v_4, v_5), (v_5, v_6), (v_1, v_5)$ of G are ranked with 4, 5, 6, 7, 8 and 9, respectively as shown in Figure 4.2(f).

4.2 Approximation Ratio

Now we will find the approximation ratio which is the measure of the goodness of our proposed approximation solution in comparison with the optimal solution of the problem. To calculate the approximation ratio we first find the lower bound of the optimal edge-ranking number of series-parallel graphs in Lemma 4.2.1 and then the upper bound of the approximate edge-ranking number of series-parallel graphs used by our approximation algorithm, thereby the approximation ratio in Lemma 4.2.2.

Lemma 4.2.1 *The optimal edge-ranking number $r'(G)$ of a series-parallel graph G satisfies $r'(G) \geq \log_2 n$, where n is the number of vertices in G .*

Proof. Solving edge-ranking problem on a graph G is equivalent to finding minimum height 1-edge-separator tree of G . Let us consider an example of a simple

series-parallel graph G composed with only series connections as shown in Figure 4.3. It is possible to construct a 1-edge-separator-tree T of G which is a balanced complete binary tree as shown in Figure 4.3(b). T is the minimum height 1-edge-separator tree among all possible 1-edge-separator trees for this graph. Now edges in nodes at the same level of T can be ranked with the same rank. So the number of ranks required for ranking the edges in G equals to $h(T) + 1$, where $h(T)$ is the height of the minimum height 1-edge-separator tree T .

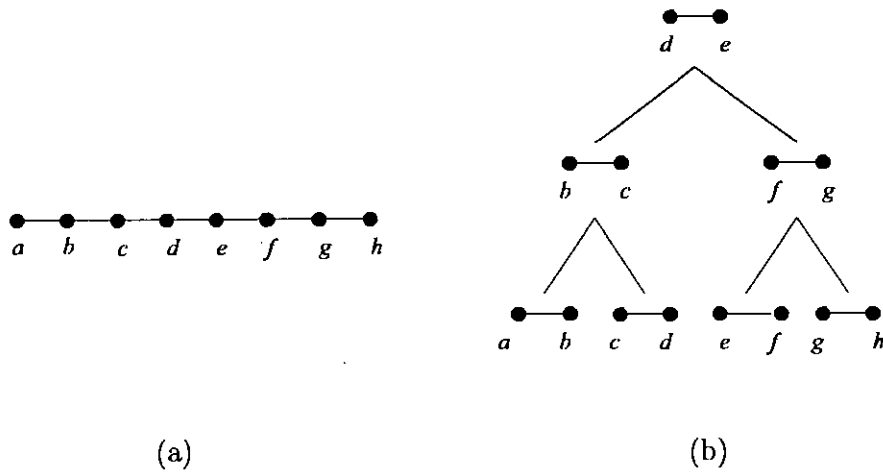


Figure 4.3: (a) A series-parallel graph G with only series connection and (b) its minimum height 1-edge-separator tree T .

The number of ranks increases with the height of the tree. Among all possible 1-edge-separator trees of a series-parallel graph complete binary tree (if possible to construct) is the tree with minimum height. We know that removing a single edge (cut edge) can result in maximum two components. In case of a 1-edge-separator tree every node cannot contain more than one edge and for the complete binary tree removing every edge in every node will result in two components. There are some series-parallel graphs for which it is not possible to construct a 1-edge-separator tree with complete binary tree structure as shown in Figure 4.4.

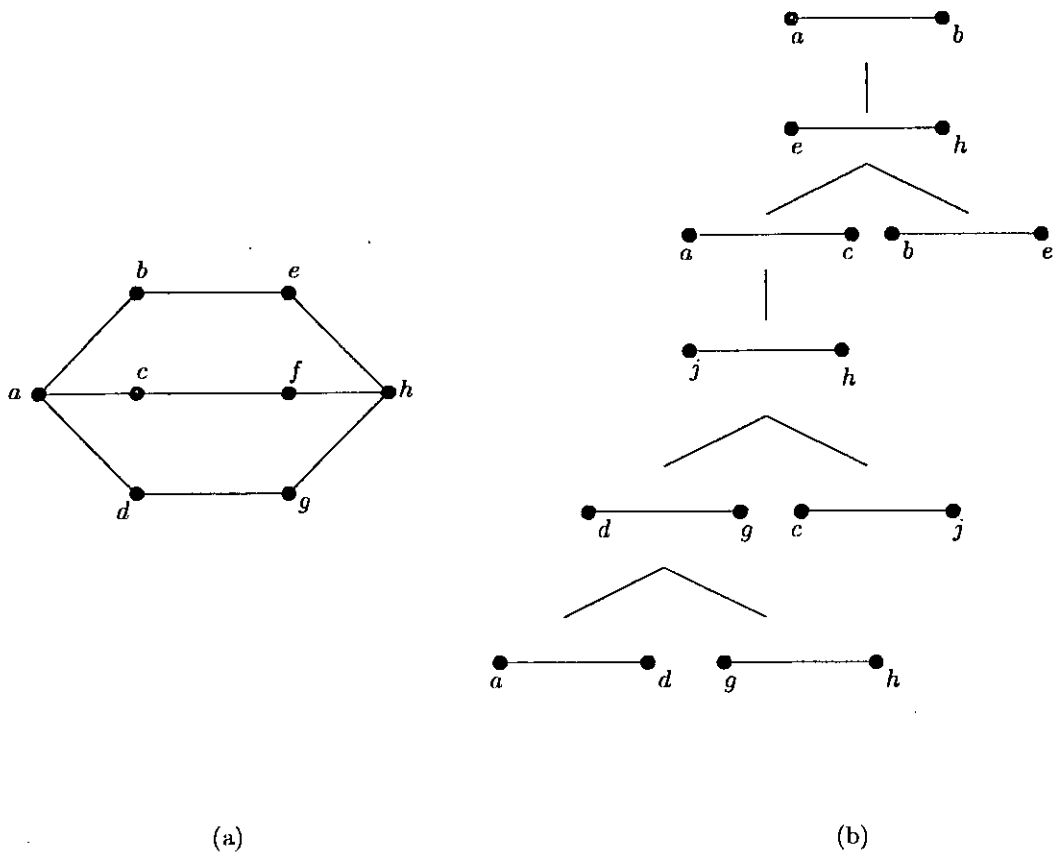


Figure 4.4: (a) A series-parallel graph G and (b) its minimum height 1-edge-separator tree T .

Now $2^{h(T)+1} - 1 = m$, where m is the number of edges in a series-parallel graph G , when T is a complete binary tree and $2^{h(T)+1} - 1 > m$ when T is not a complete binary tree. So $h(T) + 1 \geq \log_2(m + 1)$. Now in a series-parallel graph G , $m \geq n - 1$, where n is the number of vertices in G . Thus we have, $r'(G) = h_b + 1 \geq \log_2 n$.

So the smallest height possible for a 1-edge-separator tree of a series-parallel graph is that of the complete binary tree and the optimal edge-ranking number $r'(G)$ of a series-parallel graph G satisfies $r'(G) \geq \log_2 n$. \square

Lemma 4.2.2 *The Approximation algorithm SP_Approx_Rank has a ratio bound of*

$2\Delta(h + 1)/\log_2 n$, where Δ is the maximum vertex degree in G , h is the height of the 2-vertex-separator tree and n is the number of vertices in G .

Proof. Since T is a 2-vertex-separator tree of G , the number of vertices of G associated with each node x of T can be at most two. The edges in F_x (the set of edges connecting vertices in X_x to vertices in $V_{y_i}, 1 \leq i \leq d$, and also edges between vertices in X_x) require at most 2Δ ranks as there can be at most 2Δ edges in F_x . Again these edges have ranks different from the ranks used in the edges of the subgraph $\{E_{y_i}\}$ associated with the child nodes y_i of x . The algorithm can use same rank for edges in different F_x when the nodes x are in the same level of T . Since h is the height of the 2-vertex-separator tree, SP_Approx_Rank requires at most $2\Delta(h + 1)$ ranks for an edge-ranking. By Lemma 4.2.1 the lower bound for optimal edge-ranking number is $\log_2 n$. Thus SP_Approx_Rank has a ratio bound of $2\Delta(h + 1)/\log_2 n$. \square

4.2.1 Deviation from Optimality

SP_Approx_Rank has an approximation ratio of $2\Delta(h + 1)/\log_2 n$. Finding the optimal edge-ranking is equivalent to finding the minimum height 1-edge-separator tree T_e as shown in Figure 4.5. But an approximate edge-ranking of the same graph is obtained using the 2-vertex-separator tree T_v as shown in Figure 4.6. Actually this is the main reason for deviating from optimality of our approximation algorithm and it is not possible to directly compare edge-separator tree and vertex-separator tree. In the 1-edge-separator tree at each node there is only 1 edge, so 1 rank is required for each level of T_e . But in case of the 2-vertex-separator tree at each node there can be 2 vertices, so at most 2Δ edges and hence 2Δ ranks may be required at each level of T_v . Although it may happen that height of T_v is sometimes smaller than that of T_e . So the deviation from optimality depends on the height of T_v which

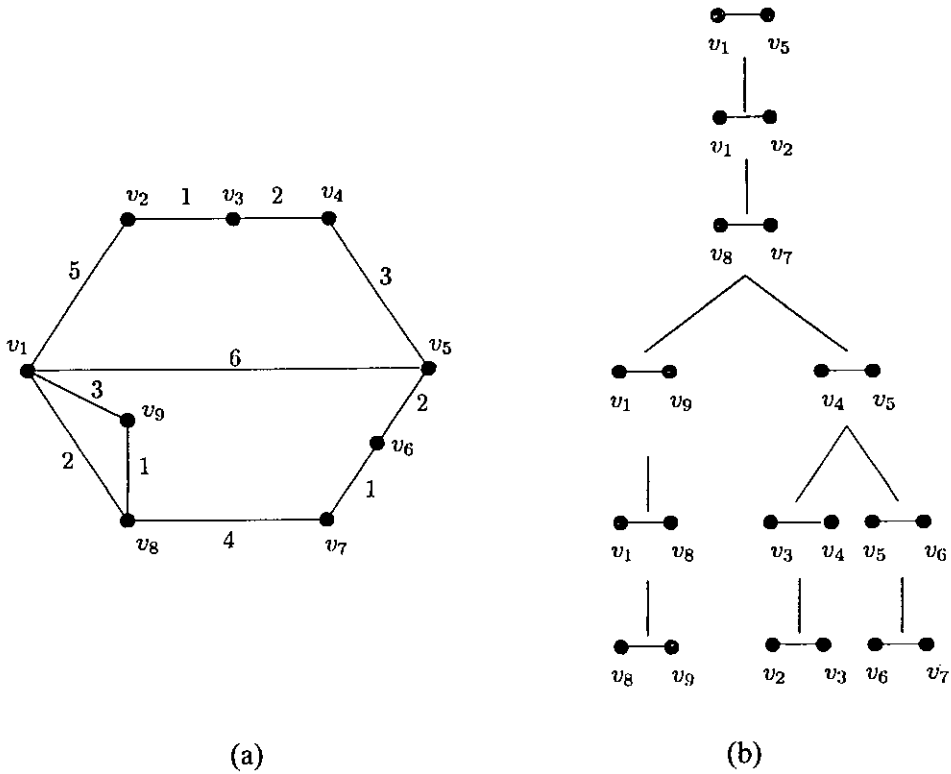


Figure 4.5: (a) The optimal edge-ranking of a series-parallel graph G , and (b) its minimum height 1-edge-separator tree T_e .

is actually h and how many edges are to be ranked for each node of T_v , which can be at most 2Δ .

4.3 Conclusion

In this chapter we present a linear-time approximation algorithm for finding the edge-ranking of a series-parallel graph. The approximation algorithm has a ratio bound of $2\Delta(h + 1)/\log_2 n$. This is the first time that an approximation algorithm is proposed for solving the edge-ranking problem on series-parallel graphs. The edge-ranking problem is \mathcal{NP} -complete for series-parallel graphs, that is, finding a

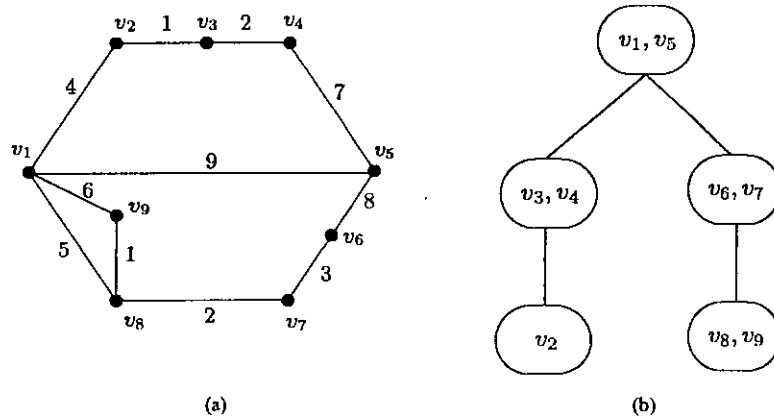


Figure 4.6: (a) The approximate edge-ranking of a series-parallel graph G , and (b) its 2-vertex-separator tree T .

polynomial-time algorithm for solving the edge-ranking problem on series-parallel graphs with unbounded maximum degree is unlikely. But our proposed linear-time algorithm can return near optimal solution.

The lower bound of the optimal edge-ranking number $r'(G)$ of a series-parallel graph G satisfies $r'(G) \geq \log_2 n$, where n is the number of vertices in G , but the upper bound of $r'(G)$ is $2\Delta \log_2 n$ [15]. The upper bound of the approximate edge-ranking number of a series-parallel graph obtained by our algorithm is $2\Delta(h + 1)$. If the height h of the 2-vertex-separator tree could be obtained close to $\log_2 n$, then the approximate edge-ranking number obtained by our algorithm would be near to optimal-edge-ranking number.

Chapter 5

Conclusion

In this thesis, we deal with the problem of finding the optimal edge-ranking of series-parallel graphs. We present a linear-time algorithm to find the 2-vertex-separator tree of series-parallel graphs and a linear-time approximation algorithm for finding the edge-ranking of series-parallel graphs using the 2-vertex-separator tree with an approximation ratio of $2\Delta(h+1)/\log_2 n$, where Δ is the maximum vertex degree in G , h is the height of the 2-vertex-separator tree and n is the number of vertices in G . The upper bound of optimal edge-ranking number of a series-parallel graph is $2\Delta \log_2 n$. If the height h of the 2-vertex-separator tree could be obtained close to $\log_2 n$, then the approximate edge-ranking number obtained by our algorithm would be near to optimal-edge-ranking number. This is the first time that an approximation algorithm is proposed for solving edge-ranking problem on series-parallel graphs.

Obtaining the 2-vertex-separator tree improves the running time of the known best algorithm for finding the optimal vertex-ranking of series-parallel graphs. Since a series-parallel graph is a partial 2-tree, it is known that each series-parallel graph has a 3-vertex-separator tree. Since we show how to construct a 2-vertex-separator-

tree of a series-parallel graph using its special structure, the upper bound of the optimal vertex-ranking number is improved and hence the running time of the best known algorithm is also improved. The optimal vertex-ranking number of a series-parallel graph is $\leq a \log_2 n$, where $a = 3$ when 3-vertex-separator tree is used and $a = 2$ when 2-vertex-separator tree is used. Using 3-vertex-separator tree the upper bound of the optimal vertex-ranking number is $3 \log_2 n$. Using 2-vertex-separator tree the upper bound of optimal vertex-ranking number improves to $2 \log_2 n$. Since the running time of known best algorithm for solving the vertex-ranking problem on series-parallel graphs depends on the upper bound of the optimal vertex-ranking number, the running time is also improved. The running time of the algorithm for solving the vertex-ranking problem on series-parallel graphs is $O(n^{2a+1} \log_2^7 n)$. So using 3-vertex-separator tree the running time is $O(n^7 \log_2^7 n)$. If we use 2-vertex-separator tree, the running time improves to $O(n^5 \log_2^7 n)$.

In Chapter 1, we focus on the background history and related motivations on this research field. We also define our problem and discuss our motivations behind solving the problem. In Section 1.1, we discuss the historical background and results on graph coloring and graph-ranking problem. Section 1.2 represents the present state of the problem and Section 1.3 deals with the scope of this thesis. At last, in Section 1.4, we discuss the results obtained for solving the problems of this thesis and compare our results with the previously achieved ones.

In Chapter 2, we discuss the required definitions for solving the problem and developing the properties. In this chapter we also mention different types of characterization, which are needed in the way of evolution. In Section 2.1, we start by giving the definitions of some basic terms of graph which are related to and used through out this thesis. Section 2.2 defines a special type of graph, series-parallel graph. It also introduces different properties of a series-parallel graph and representation of series-parallel graph through the binary decomposition tree.

Section 2.3 discusses complexity classes of the algorithm. Finally in Section 2.4 we define approximation algorithm and the approximation ratio.

In Chapter 3, we design an algorithm for finding a 2-vertex-separator tree of a series-parallel graph. In Section 3.1 we show that a series-parallel graph has a 2-vertex-separator tree. A series-parallel graph can be disconnected by removing at most two vertices. However, disconnected components that do not have the series-parallel structure may be yielded by this process. So we show that every such component has at least one cut-vertex. This immediately proves that a series-parallel graph has a 2-vertex-separator tree. Next in Section 3.2, we describe an algorithm for constructing a 2-vertex-separator tree of a simple series-parallel graph. Here we also analyze the complexity of the algorithm.

In Chapter 4, we present an approximation algorithm for solving the edge-ranking problem on a simple series-parallel graph using the 2-vertex-separator. Section 4.1 presents the algorithm, its correctness and complexity analysis. In Section 4.2 we calculate the approximation ratio of our proposed algorithm. To do that we first find the lower bound of the optimal edge-ranking number of series-parallel graphs and also the upper bound of approximate edge-ranking number of series-parallel graphs.

We first introduce the trend of solving edge-ranking problem using vertex-separator tree instead of using edge-separator tree. The following problems related to the approximation algorithm for solving the edge-ranking problem of series-parallel graphs are still open.

1. Develop a linear-time algorithm for finding the minimum height of 2-vertex-separator tree of series-parallel graphs.
2. Develop an approximation algorithm for solving the edge-ranking problem on series-parallel graphs with better approximation ratio.

3. Develop an approximation algorithm for solving the edge-ranking problem on partial k -trees.



Bibliography

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, *Addison-Wesley, Reading, MA*, 1974.
- [2] M. Bodirsky, O. Giménez, M. Kang, and M. Noy, On the number of series parallel and outerplanar graphs, *Proceedings of Discrete Mathematics and Theoretical Computer Science (DMTCS)*, (2005), pp. 383-388.
- [3] H.L. Bodlaender, Polynomial algorithms for graph isomorphism and chromatic index on partial k -trees, *Journal of Algorithms II*, (1990), pp. 631-643.
- [4] H.L. Bodlaender, J.S. Deogun, K. Jansen, T. Kloks, D. Kratsch, H. Müller, and Zs. Tuza, Rankings of graphs, *Society for Industrial and Applied Mathematics (SIAM) Journal on Discrete Math.* 21 (1998), pp. 168-181.
- [5] H.L. Bodlaender, J.R. Gilbert, H. Hafsteinsson, and T. Kloks, Approximating treewidth, pathwidth and minimum elimination tree height, *Journal of Algorithms*, 18 (1995), pp. 238-255.
- [6] J.S. Deogun, T. Kloks, D. Kratsch, and H. Müller, On vertex ranking for permutation and other graphs, *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Springer-Verlag*, 775 (1994), pp. 747-758.
- [7] J. S. Deogun, and Y. Peng, Edge ranking of trees, *Congressus Numerantium*, 79 (1990), pp. 19-28.

- [8] I.S. Duff, and J.K. Reid, The multifrontal solution of indefinite sparse symmetric linear equations, *Association for Computing Machinery (ACM) Transactions on Mathematical software*, 9 (1983), pp. 302-325.
- [9] A.V. Iyer, H.D. Ratliff, and G. Vijayan, Optimal node ranking of trees, *Information Processing Letters*, 28 (1988), pp. 225-229.
- [10] A.V. Iyer, H.D. Ratliff, and G. Vijayan, Parallel assembly of modular products - an analysis, *Technical Report Planning and Design Resource Center, Technical Report 88-06*, Georgia Institute of Technology, 1988.
- [11] A.V. Iyer, H.D. Ratliff, and G. Vijayan, On an edge-ranking problem of trees and graphs, *Discrete Applied Mathematics*, 30(1991), pp. 43-52.
- [12] M. A. Kashem, and M. E. Haque, Edge-ranking problem is NP-complete for series-parallel graphs, *Proceedings of the 4th International Conference on Computer and Information Technology (ICCIT)*, 2001, pp. 108-112.
- [13] M. A. Kashem, X. Zhou, and T. Nishizeki, Algorithms for generalized edge-rankings of partial k -trees with bounded maximum degree, *Proceedings of the 1st International Conference on Computer and Information Technology (ICCIT)*, 1998, pp.45-51.
- [14] M.A. Kashem, X. Zhou, and T. Nishizeki, Algorithms for generalized vertex-rankings of partial k -trees, *Theoretical Computer Science*, 240(2000), pp. 407-420.
- [15] M. A. Kashem, X. Zhou, and T. Nishizeki, Optimal c -vertex-rankings of series-parallel graphs, Manuscript in preparation.
- [16] T. Kloks, H. Müller, and C.K. Wong, Vertex ranking of asteroidal triple-free graphs, *Proceedings Of the 7th International Symposium on Algorithms and Computation (ISAAC'96), Lecture Notes in Computer Science, Springer-Verlag*, 1178 (1996), pp. 174-182.
- [17] T. W. Lam, and F. L. Yue, Edge Ranking of graphs is hard, *Discrete Applied Mathematics*, 85(1998), pp. 71-86.

- [18] T. W. Lam, and F. L. Yue, Optimal edge ranking of trees in linear time, *Algorithmica*, 30(2001), pp. 12-33.
- [19] C.E. Leiserson, Area-efficient, graph layouts for VLSI, *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, 1980, pp. 270-281.
- [20] J.W.H. Liu, The role of elimination trees in sparse factorization, *Society for Industrial and Applied Mathematics (SIAM) Journal of Matrix Analysis and Applications*, 11 (1990), pp. 134-172.
- [21] N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, *Journal of the Association for Computing Machinery (ACM)*, 30(1983), pp. 852-865.
- [22] M.A.H. Newton, and M.A. Kashem, An efficient algorithm for optimal vertex-ranking of permutation graphs, *Proceedings of the 2nd International Conference on Computer and Information Technology (ICCIT)*, 1999, pp. 315-320.
- [23] A. Pothen, The complexity of optimal elimination trees, *Technical Report CS-88-13*, Pennsylvania State University, USA, 1988.
- [24] A. Sen, H. Deng, and S. Guha, On a graph partition problem with application to VLSI layout, *Information Processing Letters*, 43 (1992), pp. 87-94.
- [25] K. Takamizawa, T. Nishizeki, and N. Sato, Linear time computability of combinatorial problems on series-parallel graphs, *Journal of the Association for Computing Machinery (ACM)*, 29(1982), pp. 623-641.
- [26] P. de la Torre, R. Greenlaw, and A.A. Schäffer, Optimal edge ranking of trees in polynomial time, *Algorithmica*, 13 (1995), pp. 592-618.
- [27] X.Zhou, M. A. Kashem, and T. Nishizeki, Generalized edge-rankings of trees, *The Institute of Electronics, Information and Communication Engineers (IEICE) Transactions on Fundamentals of Electronics, Communications and Computer Science*, 81-A-2(1998), pp. 310-320.



Index

- $G[E_H]$, 15
- $G[V_H]$, 15
- T , 18
- T_b , 22
- Δ , 14
- \mathcal{NP} , 25
- \mathcal{NP} -Complete, 25
- \mathcal{NP} -hard, 25
- \mathcal{P} , 25
- c -edge-ranking, 8
- c -edge-ranking number, 8
- c -edge-ranking problem, 8
- k -tree, 19
- τ , 47
- $\tau(G)$, 4
- $\tau'(G)$, 6

- algorithm, 24
- approximation algorithm, 26
- approximation ratio, 26

- binary decomposition tree, 22
 - p -node, 24
 - s -node, 24

- clique, 16

- complexity
 - linear-time, 25
 - polynomial-time, 25

- component, 17
- vertex, 30

- cycle, 16

- decomposition, 30
 - parallel, 29
 - series, 29

- edge-coloring problem, 3
- edge-ranking, 6
- edge-ranking number, 6
- edge-ranking problem, 6

- finite, 14

- forest, 18

- graph
 - adjacent, 13
 - complete, 16
 - connected, 17
 - degree, 14
 - disconnected, 17
 - incident, 14

- loop, 14
- multigraph, 14
- multiple, 14
- neighbor, 14
- parallel, 14
- simple, 14
- optimal c -edge-ranking, 8
- optimal vertex-ranking, 4
- partial k -tree, 19
- path, 16
- postorder, 48
- preorder, 34
- rank, 4
- separator, 17
 - separator, 17
 - edge, 20
 - edge separator, 17
 - vertex, 20
- separator tree, 20
- series-parallel graph, 21
 - parallel, 22
 - series, 22
- SP_Approx_Rank, 48
- subgraph, 14
 - connected, 17
 - maximal, 17
- trail, 16
- tree, 18
 - binary, 18
 - child, 18
 - height, 19
 - internal, 18
 - leaf, 18
 - node, 18
 - depth, 19
 - height, 19
 - level, 19
 - parent, 18
 - root, 18
 - rooted, 18
 - subtree, 18
- tree width, 20
- tree-decomposition, 19
- vertex-coloring problem, 3
- vertex-ranking, 4
- vertex-ranking number, 4
- vertex-ranking problem, 4
- walk, 16

