INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Partitioning Planar Graphs with Costs and Weights

By
Hua Guo

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Science
Information and System Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

February 28, 2002

© Copyright
2002, Hua Guo

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72087-X

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

Partitioning Planar Graphs with Costs and Weights

submitted by

Hua Guo

Dr. Frank Dehne
(Director, School of Computer Science)

Dr. Anil Maheshwari
(Thesis Supervisor)

Carleton University
February 28, 2002

Abstract

A good separator leads to good partitioning. A graph separator is a set of vertices or edges whose removal divides an input graph into components of bounded size. In particular, we consider connected planar graphs with costs and weights on vertices, where weights are used to estimate the sizes of the components and costs are used to estimate the quality of the separator. In this thesis, we describe a new algorithm for computing vertex separators in planar graphs. Also, we introduce a tuning technique that results in considerable reduction of the cost of the produced separators. As an application, we design an algorithm to construct an edge separator. More importantly, we show new theorems of vertex and edge separators that have theoretically guaranteed upper bounds both on the quality of the partitions and on the time needed to find them. We show that our partitioning algorithms are efficient, effective, and easy to implement.

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor, Dr. Anil Maheshwari for his guidance and support during this research.

I would also like to thank the visiting professor, Dr. Lyudmil Aleksandrov for his advise, ideas, suggestions, and comments throughout the development of this thesis. His cooperation and guidance are highly appreciated.

Also, I would like to thank my defense committee: Dr. T. Kunz from Department of Systems and Computer Engineering, Dr. J. Oommen from School of Computer Science, and the chair Dr. M. Smid from School of Computer Science, for taking their time reading my thesis, and for their corrections and comments.

Besides, I would like to thank all members of the PARADIGM group at Carleton University for their ideas and suggestions.

Last but not least, I would like to dedicate this thesis to my parents and my husband who have always given me strength, encouragement, inspiration, and support.

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Motivation	1
1.2 Scope of the Research	3
1.3 Organization of the Thesis	4
2 Background on Graph Separation	6
2.1 Preliminaries	6
2.2 Related Work	9
2.2.1 Key Results on Planar Graph Separation	10
2.2.1.1 Lipton and Tarjan's Planar Separator Theorem	11
2.2.1.2 Djidjev's Contribution	12
2.2.2 Graph Separator Theorems	13
2.2.3 Graph Partitioning Algorithms with Heuristics	14
2.2.3.1 Geometric Mesh Partitioning	15
2.2.3.2 Kernighan and Lin Algorithm	16
2.2.3.3 Recursive Spectral Bisection	17
2.2.3.4 METIS	18
2.2.4 Our Contributions	19
2.3 Previous Results	19
2.3.1 Aleksandrov and Djidjev's Fundamental Cycle Algorithm	20

2.3.2	Breadth-first Search and Single Source Shortest Path	24
2.3.3	Heaps	28
2.3.3.1	Binary Heap and Heap Property	29
2.3.3.2	Binomial Tree, Binomial Heap, and Fibonacci Heap .	31
2.3.4	Bin Packing	33
3	Partition Planar Graphs with Costs and Weights	35
3.1	Outline of Our Approach	36
3.2	Construction of t -separator	37
3.2.1	Partitioning by Fundamental Cycles	37
3.2.2	Partitioning by Levels	39
3.2.3	Packing	45
4	Reducing the Cost of Vertex t-Separator	47
4.1	Reformulating the Problem as a Shortest Path Problem	48
4.2	Efficient Computation of Shortest Paths in H	49
5	Applications	56
5.1	Variations of Vertex Separator Theorem	56
5.2	Application: Construction of Edge t -separator	57
5.3	Variations of Edge Separator Theorem	60
6	Implementation and Experimental Results	61
6.1	Design Issue	61
6.1.1	Why Choose LEDA	62
6.1.2	Priority Queue Implementation for Dijkstra's SSSP	63
6.1.3	Edge t -separator and Bin Packing	64
6.2	Experiments	65
6.2.1	Goal of Experiments and System Environment	65
6.2.2	What We Implemented and Input Parameters	66
6.2.3	Experimental Results and Observations	68
6.2.4	Comparison	80

7 Conclusion and Future Work	90
7.1 Conclusion	90
7.2 Limitations and Future Work	91
Bibliography	93

Chapter 1

Introduction

This chapter presents motivation and scope of our research, and organization of this thesis.

1.1 Motivation

Many problems can be represented by means of graphs. For example communications in computer networks can be interpreted as graphs in a natural way. Routing in computer networks is basically a shortest path problem in graphs. Graph partitioning is a basic tool in the design of efficient algorithms for solving algorithmic as well as application problems. Informally, the problem consists of finding, in an input graph G , a small set of vertices or edges, called a *separator*, whose removal divides G into two or more components of roughly equal sizes. A variety of algorithms for solving combinatorial problems are based on the divide-and-conquer strategy combined with an efficient use of graph separation on planar graphs. Such algorithms include the shortest path problem for planar or near-planar graphs [8, 14, 30], divide-and-conquer algorithms in computational geometry [15], efficient parallel factorization of sparse matrices [17, 18], problems in computational complexity [39], and algorithms for finding a VLSI layout design that minimizes the chip area [3, 37]. In scientific

computing, graph separation is used to find a distribution of the data among the processors in a high performance computing system that balances the load and minimizes the interprocessor communication.

To the author's interest, one of the applications of graph partitioning is in geographical information systems (GISs). A common data structures used in many GISs is the *triangular irregular network* (TIN), a triangulation of a point set representing a geographical region. A small sample graph in TIN format is shown in Figure 1.1. Many problems on TINs translate to solving problems on planar graphs, and hence graph partitioning is one of the main tools in designing efficient solutions for such problems.

Figure 1.1: A sample TIN graph

The goal of this work is to develop and do experiments with techniques that increase the speed of graph partitioning algorithms and/or improve the quality of the partitioning.

1.2 Scope of the Research

This thesis is focused on the study of partitioning planar graphs with costs and weights on vertices. Let $G = (V, E)$ be an n -vertex connected planar graph with real-valued positive weights $w(v)$ and costs $c(v)$ associated with its vertices. Let t be a real in $(0, 1)$. A set of vertices (edges) S is called a t -separator if their removal from G leaves no component of weight exceeding $tw(G)$. Figure 1.2 shows an example. *Cost*

Figure 1.2: Vertex separator and edge separator: (a) graph G and its vertex separator S shown in bold. (b) removal of vertices in S and their adjacent edges partitions G into 2 pieces. (c) graph G and its edge separator ES shown in bold. (d) removal of ES partitions G into 2 pieces.

of t -separator is the sum of the costs of vertices (edges) in the separator. Another requirement in terms of the quality of such separator is the cost of a separator need to be as small as possible.

In this thesis, we introduce a new approach for constructing such a vertex t -separator of G and design an algorithm for it. Besides, we develop a technique, namely “cost-reducing technique”, to reduce the cost of separator. Furthermore, we design an algorithm to obtain an edge t -separator from vertex t -separator. Our theorems of vertex t -separator and edge t -separator, developed in this research, have

theoretically guaranteed upper bounds both on the quality of the partitions and on the time needed to find them. The cost (size) of the separator is asymptotically optimal for the class of planar graphs.

We have implemented the algorithm for constructing a vertex t -separator of G where (1) costs and weights are 1.0, (2) costs are 1.0, and weights are random numbers in range $[1.0, 100.0]$, (3) costs are random numbers in range $[1, 100]$ and weights are random numbers in range $[1.0, 100.0]$. We have also implemented the algorithm for constructing an edge t -separator of G where the cost of a vertex is its degree (it is the number of edges incident to a vertex), and the weight of a vertex is either 1.0 or a random number in the range $[1.0, 100.0]$. We have conducted extensive experiments on different kinds of planar graphs to test these algorithms in terms of the quality of the separators and the time to construct them. Finally, we have made our conclusions of this research after discussing the experimental results.

In this thesis, the terms vertex separator and vertex t -separator are used interchangeably. So are the terms edge separator and edge t -separator.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows: Chapter 2 first introduces the preliminaries. Second, it presents related work in the graph separator field such as the key results of planar graph separation, results of near-planar graph separation, graph partitioning algorithms with heuristics, and our contributions. Many well-known separator theorems of planar graphs are described. At last, Chapter 2 presents previous results employed in our algorithm and implementation. This includes Aleksandrov and Djidjev's fundamental-cycle algorithm[2], breadth-first search, Dijkstra's single source shortest path algorithm, heaps, and bin packing problem.

Chapter 3 describes an algorithm on how to construct a vertex separator. It shows our vertex separator theorem for planar graphs with costs and weights on vertices. This theorem theoretically guarantees the upper bounds both on the quality of the

partitions and on the time needed to find them. The complexity of the algorithm is analyzed.

Chapter 4 introduces a cost-reducing technique for tuning our algorithm to considerably reduce the cost of the produced separator. Chapter 5 first presents how our vertex separator theorem captures many existing results on vertex separators. Then, it introduces an application of the vertex separator theorem: constructing an edge separator. Furthermore, it describes an algorithm to find edge separator from vertex separator, and introduces our edge separator theorem. At last, it shows that our edge separator theorem captures many existing results on edge separators.

Chapter 6 provides relevant details on the implementation of vertex and edge partitioning algorithms and discussion on our experimental results. It starts with the major design decisions we made for the implementation, followed by describing the goal of our experiments, the system environment, what we have implemented, and the experiments we have conducted. It ends with some experimental results, our observation and analysis.

Chapter 7 presents the conclusions and limitations of this thesis, and a short discussion on future work.

Briefly summing up, this chapter introduces the motivation of our research, the scope of our research, and the organization of this thesis.

Chapter 2

Background on Graph Separation

This chapter is focused on introducing the background on graph separation. It begins with preliminaries where notations and terminologies are introduced. Then, it presents the related work of graph separation, including the key results of planar graph separation, results of near-planar graphs, and some well-known graph partitioning algorithms with heuristics. It briefly shows our contributions to planar graph separation. At last, it reviews the previous results including algorithms and data structures that are employed in our algorithm and implementation.

2.1 Preliminaries

In this section, we introduce some notations and terminologies related to graph. A *Graph* $G = (V, E)$ consists of a set of *vertices (nodes)* and a set of *edges*, denoted by V and E , respectively. An edge is an unordered pair of distinct vertices. Vertex v and w are *adjacent* if (v, w) is an edge. An edge is *incident* to both of its *endpoints*. A *path of length k* with endpoints v_0 and v_k of G is a sequence of vertices $v_0, v_1, v_2, \dots, v_k$ such that for $1 \leq i \leq k$, (v_{i-1}, v_i) is an edge of G . If all the vertices $v_0, v_1, v_2, \dots, v_k$ are distinct, the path is *simple*. If v_0 and v_k are identical, the path is a *cycle*. The *distance* from v_0 to v_k is the length of the shortest path from v_0 to v_k . The distance

is infinite if v_0 and v_k are not connected by a path.

Graph $G_1 = (V_1, E_1)$ is a *subgraph* of graph $G_2 = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. G_1 is a *generalized subgraph* of G_2 if $V_1 \subseteq V_2$ and there is a mapping f from E_1 into the set of paths of G_2 such that for each edge $(v, w) \in E_1$, $f((v, w))$ has endpoints v and w , and no two paths $f((v_1, w_1))$ and $f((v_2, w_2))$ share a vertex except possibly an endpoint of both paths [38]. A graph is *connected* if there exists a path between any two vertices of it. The *connected components* of a graph are its maximal connected subgraphs.

Figure 2.1: A planar graph and an embedding.

A graph is said to be *embedded* in a surface S when it is drawn on S so that no two edges intersect. A graph is *planar* if it can be embedded in a plane; a *plane graph* has already been embedded in the plane. The graph in Figure 2.1(a) is planar since it is isomorphic to the plane graph in Figure 2.1(b). Kuratowski [27] proved that a graph is planar if and only if it contains neither a complete graph on five vertices (Figure 2.2(a)) nor a complete bipartite graph on two sets of three vertices (Figure 2.2(b)) as a generalized subgraph. Every graph is embeddable on some orientable surface. This can easily be seen by drawing an arbitrary graph in the plane, possibly with edges that cross each other, and then attaching a handle to the plane at each crossing and allowing one edge to go over the handle and the other under it. Figure 2.3 shows an embedding of K_5 in a plane to which one handle has been attached. The *genus* $g(G)$ of a graph G is the minimum number of handles which must be added to

 Figure 2.2: Non-planar graphs: (a) K_5 , (b) $K_{3,3}$

a sphere so that G can be embedded on the resulting surface. $g(G) = 0$ if and only if G is planar. Homeomorphic graphs have the same genus. A *finite element graph* is

 Figure 2.3: Embedding of K_5 on an orientable surface.

any graph formed from a planar embedding of a planar graph by adding all possible diagonals to each face [38]. See Figure 2.4 for an example in two dimensions.

A *tree* is a connected graph containing no cycles. Generally, a distinguished vertex, namely the *root* is specified. On a tree T rooted at vertex ρ , if vertex u is on a simple path from ρ to v , u is an *ancestor* of v and v is a *descendant* of u . In addition, if (u, v) is an edge of T , then u is the *parent* of v and v is a *child* of u . The *radius* of a

Figure 2.4: Finite element graphs: (a) a finite element graph - tapir, (b) a finite element mesh - grid.

rooted tree is the maximum distance of any vertex from the root. A *spanning tree* T of a graph G is a tree containing all the vertices of G . It partitions the edge set $E(G)$ into two sets: tree edge and non-tree edge sets. Any non-tree edge with some of the tree edges forms a unique simple cycle, called *fundamental cycle* or *T -cycle*. Figure 2.5 shows a graph and its spanning tree. Assume tree T has radius r , a fundamental cycle of T has at most $2r + 1$ vertices if containing the root of T , or at most $2r - 1$ vertices otherwise.

2.2 Related Work

Graph partitioning has been an active field of research for more than two decades. Separator theorems have been proved for both planar graphs and non-planar graphs, including graphs of bounded genus [2, 19], a separator theorem for the class of chordal graphs [20], and separator theorems for geometric graphs in two and three dimensions

Figure 2.5: A graph G and its spanning tree T rooted at ρ . Bold edges are tree edges. Edge (u, v) is a tree edge and u is the parent of v , and u is the ancestor of both v and w . Clearly, a non-tree edge forms one fundamental cycle with edges of T .

[45, 44]. Among many algorithms designed and developed for graph partitioning so far, most of them are only for *graph bisection*, i.e. vertices of a graph are separated into two halves. They are intended to be applied recursively by bisecting each half of vertices until the partitions are small and numerous enough.

The focus of our research is to develop and implement a vertex separator algorithm for any planar graph with real-valued positive costs and weights on vertices. Our algorithm have the flexibility of separating the graph into any number of pieces and it does not apply recursive bisection. It extends the results of Lipton and Tarjan[38], and Djidjev[9] and Aleksandrov[2]. In this section, we first introduce the key results of planar graph separation. Second, we review the results of near-planar graph separation. Third, we briefly introduce some key algorithms designed and developed for graph partitioning using heuristic methods.

2.2.1 Key Results on Planar Graph Separation

Planar graph partitioning is the focus of our interest. Our work extends the results of Lipton and Tarjan[38], and Djidjev[9].

2.2.1.1 Lipton and Tarjan's Planar Separator Theorem

Lipton and Tarjan were first to formally study the graph partitioning problem in [38], where they introduced the notion of a *graph separator*. They showed that in any n -vertex planar graph with nonnegative vertex weights summing to $w(G)$, there exists a separator of no more than $\sqrt{8n}$ vertices whose removal divides the graph into two components none of whose weight is more than $(2/3)w(G)$. This theorem, which we called “main” theorem of Lipton and Tarjan, forms the basis of all known separator theorems for planar graphs. They extended their main theorem to graph bisections by showing that in any n -vertex planar graph with nonnegative vertex weights summing to $w(G)$, there exists a separator of no more than $\sqrt{8n}(1 - \sqrt{2/3})$ vertices whose removal divides the graph into two components none of whose weight exceeds $(1/2)w(G)$. In addition, Lipton and Tarjan applied their theorem to finite element graphs. They showed that in any n -vertex finite element graph with nonnegative vertex costs summing to $w(G)$, suppose no element of G has more than k boundary vertices, there exists a separator of no more than $4\lfloor k/2 \rfloor \sqrt{n}$ vertices whose removal divides the graph into two components none of whose weight exceeds $(2/3)w(G)$.

Furthermore, Lipton and Tarjan described applications of the main theorem in [39]. For instance, their main theorem can be applied for an efficient algorithm for finding maximum independent sets in planar graphs, pebbling, and lower bounds on the complexity of planar Boolean circuits. The definitions of those problems are given next to be self-contained.

Maximum independent set problem is to find a maximum number of pairwise non-adjacent vertices in a planar graph. *Pebbling* is a one-person game that arises in register allocation problems[51], the conversion of recursion to iteration[47], and the study of time-space tradeoffs[48]. The game involves placing pebbles on the vertices of a directed acyclic graph G according to certain rules. A given step of the game consists of either placing a pebble on an empty vertex of G (i.e. *pebbling* the vertex) or removing a pebble from a previously pebbled vertex. A vertex may be pebbled only if all its predecessors have pebbles. The object of the game is to successively pebble

each vertex of G (in any order) subject to the constraint that at most a given number of pebbles are ever on the graph simultaneously. In application, pebbles correspond to memory units and the objective is to minimize the total memory needed in order to carry out the computation represented by G . A *Boolean circuit* is an acyclic directed graph such that each vertex has in-degree zero or two, the predecessors of each vertex are ordered, and corresponding to each vertex v of in-degree two is a binary Boolean operation b_v . With each vertex of the circuit we associate a Boolean function which the vertex computes. The circuit computes the set of functions associated with its vertices of out-degree zero (output).

2.2.1.2 Djidjev's Contribution

Djidjev improved the bound on the size of the separator to $\sqrt{6n}$ in [11] compared to $\sqrt{8n}$, the bound that Lipton and Tarjan proved. In many applications, there are costs and weights associated with vertices of a graph. For instance, cost might be used to represent the amount of data exchanged between communication of processes in a distributed system. Djidjev[9] generalized Lipton and Tarjan's results to planar graphs with costs and weights on vertices, whereas costs are used to evaluate the size of the separator and weights are used to evaluate the sizes of the components. The main theorem in [9] states that in a planar graph G with non-negative vertex weights and costs there exists a vertex set of total cost $\sqrt{8} \sqrt{\sum_{v \in V} (c(v))^2}$ whose removal leaves no component of weight exceeding $(2/3)w(G)$, where $c(v)$ denotes the cost of a vertex v . Moreover, it is shown that this bound is tight up to a constant factor, and the partition can be computed in linear time. Besides, Djidjev showed that his main theorem in [9] implied a 1/2-separator theorem with a bound on the cost of the separator increased only by a constant factor. It states that in any planar graph G with non-negative vertex weights and costs, there exists a 1/2-vertex separator of G of cost $O(\sqrt{\sum_{v \in V} (c(v))^2})$, which can be found in $O(|G|)$ time.

Djidjev [9] applied his theorem to solve graph embedding problem, pebbling problem, and multi-commodity flow problem, etc. Many problems in VLSI design and in network simulation can be formulated as graph embedding problems. In case of

planar graphs or graphs of bounded genus, one can use Djidjev's graph separation results to find embedding with good properties. The multi-commodity flow problem is related to optimally routing several different commodities in a network in order to satisfy a given set of demands. The total amount of flow through any of the edges should not exceed the capacity of that edge. Approximation algorithms exist that find whether there exists a *feasible* flow (flow that satisfies the demands and obeys all capacity constraints).

2.2.2 Graph Separator Theorems

Separator theorems have been proved for graphs of bounded genus [2, 19]. Gilbert et al.[19] introduced a graph separator theorem for graphs of bounded genus. They showed that a graph of genus g with n vertices has a set of at most $6\sqrt{gn} + 2\sqrt{2n} + 1$ vertices whose removal leaves no component with more than $2n/3$ vertices. They showed that such a separator can be found in $O(n + g)$ time for a given embedding of G in its genus surface. Then, they extended this theorem by allowing weights on vertices of G . They showed that in any n -vertex graph G of genus g whose vertices have nonnegative weights, the vertices of G can be partitioned into three sets A , B , and C such that no edges joins a vertex in A with a vertex in B , C contains $O(\sqrt{gn})$ vertices, and neither A nor B contains more than half the total weight. The set C can be found in $O(n + g)$ time. One example of this theorem is to find an approximately maximum independent set I in a graph of genus g .

Meanwhile, Aleksandrov and Djidjev [2] introduced their graph separator theorem for graphs of bounded genus. They showed that in any n -vertex graph G with nonnegative vertex weights, $I(G)$ is a 2-cell embedding of G on an orientable surface of genus g , for any $t \in (0, 1)$ there exists a t -separator of G of no more than $4\sqrt{(g + 1/t)n}$ vertices. The separator can be found in $O(n + g)$ time, given $I(G)$.

A separator theorem has been proved for the class of chordal graphs [20]. Gilbert et al.[20] found that a chordal graph with n vertices and m edges can be cut into half by removing $O(\sqrt{m})$ vertices. A *chordal* graph is an undirected graph if every

cycle of length at least four has a *chord*, which is an edge joining two vertices that are not adjacent on the cycle. Chordal graphs have also been called triangulated graphs, monotone transitive graphs, rigid circuit graphs, and perfect elimination graphs. A *clique* of a graph is its maximal complete subgraph [28]. Figure 2.6 gives an example of clique. Gilbert et al. proved that a chordal graph G has n vertices and m edges, and with p vertices in its largest clique, suppose nonnegative weight are on vertices and the sum of the weights is n , G contains a clique whose removal leaves no connected component of weight more than $n/2$. Unless $n = 1$, the clique can be chosen to have at most $p - 1$ vertices.

Figure 2.6: (a), (b), (c), and (d) show a clique as a double-line triangle.

Other kinds of graphs that can be partitioned evenly by deleting $o(n)$ vertices are trees ($O(1)$ vertices [22, 34]), outerplanar graphs ($O(1)$ vertices [33]), hypercubes ($O(n/\sqrt{\log n})$ vertices [21]), etc. Also, separator theorems have been proved for geometric graphs in two and three dimensions [45, 44].

2.2.3 Graph Partitioning Algorithms with Heuristics

Finding optimal solution to graph partitioning is NP-hard, and finding good graph separators in practice can be very expensive. A number of researchers, mainly working in the scientific computing area, have designed and implemented graph partitioning algorithms which are based on heuristics, for instance, the Kernighan-Lin method [36], genetic algorithms[40], or combinations of different methods (see [4] for a survey of results).

Informally, there are two classes of partitioning heuristics, depending on the information available about the graph, for instance coordinate information. For graphs with coordinate information, the partitioning heuristics are *geometric* in the sense that nodes of graphs lie in the plane, then the algorithms seek either a line or a circle that divide the nodes in half, with (about) half the nodes on one side of the line or circle, and the rest on the other side. All these algorithms work by trying to find a geometric *middle* of the nodes and then pass a line (or plane) or a circle (or sphere) through this middle point. Note that these algorithms completely ignore where the edges are and implicitly assume the edges connect nearby nodes only. Geometric Mesh Partitioning by Gilbert et al.[16] is in this category. On the other hand, graphs without coordinate information require combinatorial rather than geometric algorithms to partition them. Some well-known algorithms such as Kernighan-Lin algorithm [36], Recursive-spectral Bisection [46], and METIS [49] are in this category. In the rest of this subsection, we're going to review Geometric Mesh Partitioning by Gilbert et al, Kernighan-Lin algorithm, Recursive Spectral Bisection, and METIS by Karypis and Kumar, etc.

2.2.3.1 Geometric Mesh Partitioning

Gilbert et al.[16] introduced a Matlab implementation¹ of a partitioning algorithm, namely Geometric Mesh Partitioning, for geometric graphs (graphs whose embedding in two or three dimensions satisfies certain requirements) based on the theoretical work of Miller, Teng, Thurston, and Vavasis [42, 43], who showed that certain classes of “well-shaped”² finite element meshes have good separators. The algorithm partitions a d -dimensional mesh by finding a suitable sphere in d -space, and dividing the vertices into those interior and exterior to the sphere. The cutting sphere is found by a randomized algorithm that involves a conformal mapping of the points on the surface of a sphere in $(d + 1)$ -space. The algorithm is simplified as follows:

¹Matlab implementation of Geometric Mesh Partitioning is available electronically via anonymous FTP from <ftp:parc.xerox.com> at </pub/gilbert/meshpart.uu>

²“Well shaped” mesh means that for instance, a mesh having bounded aspect ratio or having angles that are not too small or too large.

Geometric Mesh Partition Algorithm

Input - The (x,y) coordinates for every vertex in the graph

Output - The vertices in the original graph are divided into three sets A , B , and C , where A and B don't share an edge and C is a separator whose removal disconnects all paths from A to B .

Algorithm

1. Map the points from R^2 onto a unit sphere in R^3 .
2. Determine the center-point³ for the points in R^3 .
3. Rotate and scale the points so that the center-point is at the origin.
4. Find a cutting plane that goes through the center-point.
5. Project down to R^2 the points and cutting plane from step 4.
6. Determine sets A , B , and the separator C

Gilbert et al.[16] claimed that first, though the theory behind the geometric partitions is fairly complicated, the algorithms themselves are quite simple and easy to implement. Second, the implementation can be made quite efficient. Third, the produced partitions are good.

2.2.3.2 Kernighan and Lin Algorithm

The idea of Kernighan and Lin algorithm [36] is to partition the vertex set of a graph into two groups A and B . Within both groups, then look for sequences of swaps of vertices that reduce the number of edges to be removed. Figure 2.7 shows an example.

³Point q is a *Radon point* [13] of a set P of points in R^d if P can be partitioned into two disjoint subsets P_1 and P_2 such that q lies in the intersection of the convex hull of P_1 and the convex hull of P_2 . The idea of the center-point heuristic is to repeatedly replace randomly chosen groups of $d + 2$ points with their Radon points. Eventually the set is reduced to a single point, which is the approximate center-point.

Figure 2.7(a) shows a graph abstracted as a circle and a separator represented by the curly vertical line in the middle. The separator partitions the vertex set into two groups: A and B . A_1 and B_1 are identified as subgroups of A and B , respectively. Swapping vertices in A_1 and B_1 , i.e. A_1 becomes a subgroup of B and B_1 becomes a subgroup of A , will reduce the number of edges to be removed. Figure 2.7(b) shows the resulting separator after swapping A_1 and B_1 .

Figure 2.7: Illustrate Kernighan and Lin Algorithm: (a) a graph is abstracted as a circle and a separator, represented by the curly vertical line in the middle, partitions the graph into A and B . Identify A_1 and B_1 as subgroups of A and B , respectively. Swapping A_1 and B_1 , i.e. A_1 becomes a subgroup of B and B_1 becomes a subgroup of A , will reduce the number of edges to be removed. (b) the resulting separator after swapping A_1 and B_1 .

2.2.3.3 Recursive Spectral Bisection

Pothen et al.[46] proposed one of the most successful heuristic technique - Recursive spectral bisection (RSB) - for finding a minimum cut graph bisection. This method has been widely used in practice [50, 52] especially for unstructured finite element meshes. The general form of a class of these algorithms is described here:

Function Recursive Bisection

Input - A graph $G = (V, E)$ and $p =$ number of groups($p = 2^n$)

1. Find an optimal bisection (spectral bisection), G' and G'' of G
2. While ($|G'| > |V|/p$)
 - (a) Perform Recursive Bisection(G')
 - (b) Perform Recursive Bisection(G'')
3. Return the subgraphs $G_1, G_2, \dots, G_p$

2.2.3.4 METIS

In spite of the extensive algorithmic work on constructing separators or using them in the design of other combinatorial algorithms, there is very little work on the efficient implementation of partitioning algorithms. Software packages containing implementations of such type of algorithms include Chaco [31]⁴, and METIS [49]. METIS⁵ is the name of the software implementation of the methodology of Karypis and Kumar [25, 23, 26]. It is a family of programs. One of the key purposes of METIS is for partitioning unstructured graphs and hyper-graphs. A high-level view of the methodology is the following:

1. Collapse the original graph down to a smaller graph through a series of matchings. Matched vertices are combined as a single vertex as are the corresponding set of edges that were incident to vertices.
2. Partition the reduced graph (for example, by spectral bisection)
3. Expand out the smaller graph, while maintaining the general partition created in step 2.

After step 1, one vertex may represent several vertices. If vertex v is one of the vertices in the final collapsed graph and is assigned to group 1, then all of the vertices that v

⁴Chaco is available from Bruce Hendrickson at Los Alamos National Las, bahendr@cs.sandia.gov

⁵The package METIS by George Karypis at University of Minnesota is available electronically. More information about the METIS family can be found at METIS homepage at <http://www-users.cs.umn.edu/karypis/metis/>.

represents are initially assigned to group 1. The speed of this algorithm comes from the fact that the size of the graph that is actually partitioned may be substantially smaller than the original graph. At each phase of expansion, reassignment of vertices may take place. In the actual implementation of METIS, the authors implement four different algorithms for partitioning a graph. Three of the algorithms are based on graph growing heuristics. The other algorithm is based on spectral bisection. The authors evaluate the performance of the different algorithms in [24].

2.2.4 Our Contributions

Although these heuristic-based algorithms usually work well in practice for the chosen applications, they can not guarantee a balanced partition in the worst case. In contrast, we aim at developing a graph partitioning algorithm which theoretically guarantees upper bounds both on the quality of the partitions and on the time needed to find them that are asymptotically optimal for the class of planar graphs. Besides, our algorithm does not follow the idea of applying graph bisection recursively. It has the flexibility of accepting *any* number of pieces to separate a graph. Moreover, a tuning technique is developed to considerably reduce the cost of the separator produced. Extensive experiments have been done to test the quality of the partitions produced and the time needed. Experimental results indicate that our tuning technique is effective and computationally efficient. They also show that our algorithm always produces quite balanced partitions and separators whose worst-case cost (size) is guaranteed. At last, our algorithm is efficient, effective, and easy to implement.

2.3 Previous Results

In this section, we survey the previous results on which this thesis is based. We will not repeat the planar graph separator theorems of Lipton and Tarjan, and of Djidjev since they are mentioned in Subsection 2.2.1. We first introduce separation graph, then review Aleksandrov and Djidjev's approach of graph separations by means

of fundamental cycles [2]. Second, we briefly review breadth-first search(BFS) and Dijkstra's Single Source Shortest Path (SSSP) algorithm. Third, we review heaps. At last, we introduce bin packing problem. These algorithms are employed in the implementation of our algorithms for constructing vertex and edge separators.

2.3.1 Aleksandrov and Djidjev's Fundamental Cycle Algorithm

Aleksandrov and Djidjev [2] defined *separation graph* of an embedded graph. Here, we simplify the definition to planar graph.

Definition 1 *A separation graph of planar graph G with respect to the spanning tree T is a graph $S = S(G, T)$ such that*

$$V(S) = F(G)$$

$$E(S) = \{(f_1, f_2) : f_1, f_2 \in F(G), f_1 \text{ and } f_2 \text{ share a non-tree edge} \}$$

where $V(S)$ is the vertex set of S , $F(G)$ is the face set of G , and $E(S)$ is the edge set of S .

Figure 2.8: (a) A triangulated graph G and a spanning tree T . Edges in T are drawn in bold. (b) Tree edges are shown as middle-thick bold line segments and separation graph $S(G, T)$ is shown with the thickest bold line segments.

Figure 2.8 presents a graph, a spanning tree and its separation graph. The nature of separation graph S of G is that there is a one-to-one mapping from a face in G to a vertex in S , and from a non-tree edge in G to an edge in S . So, an edge in S has a mapping to a fundamental cycle in G . Partitioning a planar graph by fundamental cycles originated from Jordan curve theorem [29] ⁶. Using Jordan curve theorem, Lipton and Tarjan [38] showed that for a graph G with non-negative vertex weights summing to $w(G)$ and its spanning tree T of radius r , if G is triangulated, then there is at least one fundamental cycle that contains no more than $2r + 1$ vertices whose removal partitions G into two components, none of which has weight exceeding $(2/3)w(G)$.

Aleksandrov and Djidjev [2] introduced an algorithm, presented in Algorithm 2, to find a set of fundamental cycles whose removal partitions graph G into components whose weight does not exceed $tw(G)$ where $t \in (0, 1)$, and G is a planar graph with non-negative vertex weights summing to $w(G)$. Essential to this approach is the data structure called *separation graph*, and a technique they developed for manipulating separation graphs. Separation graphs are sparse graphs of degree 3 (it is a binary tree).

For any binary tree T_b with weights on edges, let the set of edges of T_b be divided into levels $E_0, \dots, E_r$ according to their distance to some fixed vertex of T_b . For $e \in E(T_b) \setminus E_0$, denote by $pr(e)$ the unique edge at the lower level that shares a common vertex with e . Create an additional edge to be a predecessor of the edges from E_0 . Algorithm 1 finds a set of edges whose removal separates T_b into subtrees, each of which has weight less than $tw(T_b)$ where $t \in (0, 1)$ and $w(T_b)$ is the weight of T_b . Starting from the bottom, the weight of an edge at the bottom level is added to the weight of $pr(e)$. Continue to do this until some edge has weight greater than $tw(T_b)$, then that edge is put into edge separator. Figure 2.9 shows an example of Algorithm 1.

Algorithm 2 basically has three phases:

⁶Jordan curve theorem [29] says for any closed curve C in a plane, removal of C divides the plane into exactly two connected regions, the “inside” and the “outside” of C .

Figure 2.9: (a) A binary tree T_b with unit weight on edges, $t = 0.5$ and $tw(T_b) = 7.5$. (b) The weight of an edge e at the bottom level is added to the weight of edge $pr(e)$. (c) Weight of an edge shows the original weight of that edge plus the weight of the subtree below that edge. The edge with weight 4 is shown in bold. Since $4 > (t/2)w(T_b) = 3.75$, this edge is put into edge separator. (d) Another two edges whose subtree weight 4 and 7 are added into edge separator. (e) T_b is separated into 3 subtrees, none of which has weight greater than $tw(T_b) = 7.5$.

1. Construct a separation graph $S = S(G, T)$ of triangulated graph G with respect to breadth-first spanning tree T . Assign weights to the edges of S .
2. For each connected component Q in S with weight $w(Q) > tw(S)$, find a breadth-first spanning tree T_Q and insert the non-tree edges into empty set M . If the weight of the tree $w(T_Q) > tw(S)$, then apply Algorithm 1 to find a set of edges M_{t_1} whose removal partitions T_Q . Add edges in M_{t_1} into M . Set M is the edge separator of S .
3. find the set of fundamental cycles corresponding to each edge in M .

Aleksandrov and Djidjev showed that one can construct separators for G by constructing edge separators for the separation graph of G . They also showed that such a separator can be found in linear time.

Algorithm 1 *Tree – Separation*($T_b, E_0, \dots, E_r, \rho, t_1, w(\cdot)$)

```

1: Input: A binary tree  $T_b$ , the set of levels  $E_0, \dots, E_r$  with respect to a root  $\rho$ , a
   parameter  $t_1 \in (0, 1)$ , and the weights  $w(\cdot)$  on the edges of  $T_b$ .
2: Output: A set  $M_{t_1}$  which is an  $t_1$ -edge separator of  $T_b$ .
3: for  $i = r, r - 1, \dots, 0$  do
4:   for each  $e \in E_i$  do
5:     if  $w(e) > (t_1/2)w(T_b)$  then
6:       insert  $e$  in  $M_{t_1}$ 
7:     else
8:        $w(\text{pr}(e)) \leftarrow w(\text{pr}(e)) + w(e)$ 
9:     end if
10:  end for
11: end for

```

Algorithm 2 *Fundamental_Cycle*(G) – Find a set of fundamental cycles of G

```

1: Triangulate graph  $G$  so that  $G$  is of degree 3. Then find a breadth-first spanning
   tree  $T$  of  $G$  from any vertex of  $G$ .
2: Construct the separation graph  $S$  of  $G$  with respect to  $T$ .
3: Assign vertex weight of  $G$  to edges of  $S$ .
4: Find the set of connected components of  $S$ .
5: for each connected component  $Q$  with  $w(Q) > tw(S)$  do
6:   Find a breadth-first spanning tree  $T_Q$  of  $Q$  and divide the set of edges of  $T_Q$ 
   into levels with respect to the root of  $T_Q$ .
7:   Insert the set of non-tree edges of  $Q$  in edge separator  $M$ .
8:   if  $w(T_Q) > tw(S)$  then
9:     Apply Algorithm 1 on  $T_Q$  with parameter  $t_1 = tw(S)/w(T_Q)$  to find an  $t_1$ -
     edge separator  $M_{t_1}$  of  $T_Q$ . Add  $M_{t_1}$  to the set  $M$ .
10:  end if
11: end for
12: Find the set of fundamental cycles corresponding to each edge in  $M$ .

```

Later, Djidjev [9] showed the cost of such fundamental cycle when both costs and weights on vertices exist. He defined *steep spanning tree* of a planar graph and redefined *level* of it. Let T be a spanning tree of G with root ρ and for any vertex v of G let $level(v)$ denote the sum of the vertex costs on the simple tree path from v to the root ρ . In fact, $level(v)$ represents the shortest distance from v to the root. T is called *steep spanning tree* if for any vertex $w \neq \rho$ the parent of w is a vertex adjacent to w with a minimum level. Such a steep spanning tree T can be constructed in linear time by 2 steps: first, replace any edge (v, w) of G by a pair of directed edges, one is from v to w of length $c(w)$ and the other is from w to v of length $c(v)$ (See Figure 2.10). Denote the resulting graph by G' . Second, choose an arbitrary vertex ρ as the root of T . Run the linear time single source shortest path algorithm from [30] on G' with source ρ . Such a T is the steep spanning tree, which is the shortest path tree of G' .

Figure 2.10: (a) an edge (v, w) of graph G , (b) replace (v, w) by a pair of directed edges, one from v to w of length $c(w)$ and the other from w to v of length $c(v)$.

Djidjev [9] proved that in any triangulated planar graph G with non-negative real vertex costs and a steep spanning tree T of maximum level l_{max} , there is a $2/3$ -vertex separator of G that is a fundamental cycle with cost not exceeding $2l_{max} + c(\rho)$ where ρ is the root of T and $c(\rho)$ is the cost of the root. This result is used in our algorithm.

2.3.2 Breadth-first Search and Single Source Shortest Path

Breadth-first search (BFS) is one of the simplest algorithm for searching a graph. Given a graph G and a distinguished *source* vertex s , BFS systematically explores the edges of G to “discover” every vertex reachable from s . It computes the distance

from s to all such reachable vertices. For any vertex v reachable from s , the path in the breadth-first tree from s to v corresponds to a “shortest path” from s to v in G , i.e. a path containing the fewest number of edges. The algorithm works on both directed and undirected graphs.

BFS colors each vertex white, gray, or black to keep track of progress. All vertices start out white and may later become gray and then black. A vertex is *discovered* the first time it is encountered during the search, at which time it becomes nonwhite. Black vertices are those that have been discovered. A vertex adjacent to a black vertex is either gray or black. Gray vertices may have some adjacent white vertices which are not discovered yet.

BFS constructs a BFS tree, initially containing only its root, which is the source vertex s . Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex u , the vertex v and the edges (u, v) are added to the tree. The algorithm is shown in Algorithm 3. It uses a first-in, first-out queue Q to manage the set of gray vertices. Denote the predecessor or parent of a vertex u by $p[u]$, denote the distance of u from s by $d[u]$, denote the color of u by $c[u]$. Denote the vertex set of G by $V[G]$. Enqueue(Q, v) puts v into Q as the tail element, and Dequeue(Q) moves the head element out of Q . The time complexity of BFS algorithm is $O(|V| + |E|)$ since $O(|E|)$ time is spent in total scanning adjacency lists and the initialization takes $O(|V|)$ time.

Single Source Shortest Path (SSSP) Problem is described as follows: given a directed graph $G = (V, E)$, with non-negative costs on each edge, and a selected source vertex s in V , find the minimum cost path from s to every other vertex in V . Costs can be distances, times, etc. associated with edges. Dijkstra’s algorithm is known to be a good algorithm to find a shortest path. It functions by constructing a shortest-path tree from the source vertex to every other vertex in the graph. It is a greedy algorithm ⁷ for the SSSP problem.

SSSP algorithm begins by selecting any vertex of the graph to be the source vertex

⁷A “greedy” algorithm always makes the locally optimal choice under the assumption that this will lead to an optimal solution overall.

Algorithm 3 $BFS(G, s)$ – Breadth-first Search of G with Source s

```
1: for each vertex  $u \in V[G] - s$  do
2:    $c[u] \leftarrow \text{WHITE}$ 
3:    $d[u] \leftarrow +\infty$ 
4:    $p[u] \leftarrow \text{NIL}$ 
5: end for
6:  $c[s] \leftarrow \text{GRAY}$ 
7:  $d[s] \leftarrow 0$ 
8:  $p[s] \leftarrow \text{NIL}$ 
9:  $Q \leftarrow \{s\}$ 
10: while  $Q \neq \emptyset$  do
11:    $u \leftarrow$  the head element of  $Q$ 
12:   for each  $v \in u$ 's adjacency list do
13:     if  $c[v]$  is WHITE then
14:        $c[v] \leftarrow \text{GRAY}$ 
15:        $d[v] \leftarrow d[u] + 1$ 
16:        $p[v] \leftarrow u$ 
17:       Enqueue( $Q, v$ )
18:     end if
19:   end for
20:   Dequeue( $Q$ )
21:    $c[u] \leftarrow \text{BLACK}$ 
22: end while
```

s , and assigning a distance to s by 0, and a distance to all other vertices by the value of maximum infinity. Then, the algorithm puts s into an empty queue Q . When Q is not empty, extract the vertex with minimum distance (cost) to s and assign it to variable u . For any vertex v adjacent to u , if the distance of v is greater than the distance of u plus the cost from u to v , then the distance of v is updated by the new value. This operation is called “relaxation”, and it actually decreases key of v . If the distance of v is updated through u , then u is the predecessor of v . Put v in Q . This process is repeated until Q is empty which means we find a shortest (w.r.t. distance or cost) path from the source to all other vertices of the graph. Note that BFS finds shortest paths in case of all edges having unit cost.

The data structures used by Dijkstra’s algorithm include: a cost matrix C whose entry $C[u, v]$ is the cost on the edge connecting vertex u and v if such an edge exists or infinity otherwise, a set of vertices S contains all the vertices whose shortest path from the source is known, a priority queue Q , a distance vector D where an entry $D[v]$ contains the cost of the shortest path (so far) from the source to vertex v , a predecessor vector P whose entry $P[v]$ contains the predecessor vertex of v . Dijkstra’s SSSP algorithm is shown in Algorithm 4.

Theoretically, if the priority queue is implemented with binary heap, both Extract_MIN and relaxation operations (or Decrease_Key) takes time $O(\log |V|)$. There are $O(|V|)$ such Extract_Min operations, so the total Extract_Min operations take $O(|V| \log |V|)$. Since the total number of edges in all the adjacency lists is $O(|E|)$, there are a total of $O(|E|)$ relaxation operations. So, the total relaxation operations take $O(|E| \log |V|)$. Therefore, the total running time is $O((|V| + |E|) \log |V|)$, which is $O(|E| \log |V|)$ if all vertices are reachable from the source. On the other hand, if the priority queue is implemented with a Fibonacci heap, we can in fact achieve a running time of $O(|V| \log |V| + |E|)$ because the amortized cost of each of the $|V|$ Extract_Min operations is $O(\log |V|)$, and each of the $|E|$ relaxation takes only $O(1)$ amortized time.

Algorithm 4 *Dijkstra_SSSP*(G, s, C, D, P) – Dijkstra's SSSP of G with Source s

```

1: for each vertex  $u \in V[G]$  do
2:    $D[u] \leftarrow +\infty$ 
3:    $P[u] \leftarrow \text{NIL}$ 
4: end for
5:  $D[s] \leftarrow 0$ 
6:  $S \leftarrow \emptyset$ 
7:  $Q \leftarrow V$ 
8: while  $Q \neq \emptyset$  do
9:    $u \leftarrow \text{Extract\_Min}(Q)$ 
10:   $S \leftarrow S \cup \{u\}$ 
11:  for each  $v \in u$ 's adjacency list and  $v \in V \setminus S$  do
12:    if  $D[v] > D[u] + C[u, v]$  then
13:       $D[v] \leftarrow D[u] + C[u, v]$ 
14:       $P[v] \leftarrow u$ 
15:    end if
16:  end for
17: end while

```

2.3.3 Heaps

A priority queue is used in Dijkstra's SSSP algorithm. *Priority queue* is a data structure for maintaining a set S of elements, each with an associated value called *key*. It supports the following operations:

1. *Insert*(S, x): inserts the element x into the set S .
2. *Maximum*(S) or *Minimum*(S): returns the element of S with the largest or smallest key.
3. *Extract_Max*(S) or *Extract_Min*(S): removes and returns the element of S with the largest or smallest key.

Heaps are used to implement a priority queue. In the rest of this section, we introduce binary heap, binomial tree and binomial heap, and Fibonacci heap. We also compare their running time for some basic operations.

2.3.3.1 Binary Heap and Heap Property

There are several kinds of heaps, among them binary heap and Fibonacci heap are of our interest⁸. *Binary heap* data structure can be viewed as a complete binary tree: the tree is completely filled on all levels except possibly the leaves, which is filled from left to right. This binary tree satisfies *heap property*: the value of a node is at most the value of its parent. Thus the largest element in a heap is stored at the root. Such a heap is “max” binary heap. An example is shown in Figure 2.11(a). On the other hand, if a heap keeps the smallest element at the root and the value of a node is at least the value of its parent, such a heap is “min” heap. A simple and flat implementation of binary heap is an array representation which stores the keys in an array. The key of the root node is the first element of the array. The key of the root’s left and right children are the second and third elements of the array, and so on. See Figure 2.11(b).

There are four basic procedures of binary heap:

1. *Heapify*: is the key function to maintaining the heap property. For example, consider max binary heap, if a value stored at a node i is smaller than the value of its children, heapify lets the smaller value float down in the heap so that the subtree rooted node i becomes a heap. Heapify runs in $O(\log n)$ time which equals to the height of a binary tree with n nodes.
2. *Build-heap*: uses Heapify in a bottom-up manner to convert an unordered set of elements into a heap. For example, if n elements are stored in an array $A[1 \dots n]$, then the elements in the subarray $A[\lfloor n/2 \rfloor + 1 \dots n]$ are all leaves of

⁸The purpose that we introduce heap into details is that the operations of heap such as “Extract_min” plays role in the running time of our algorithm which employs heap as priority queue to implement Dijkstra’s SSSP algorithm. From theoretical point of view, Fibonacci heaps are preferable compared to binary heaps, but they are complicated in terms of data structure. From practical point of view, normally binary heaps can be implemented as an array, and this simple data structure makes binary heaps more attractive in practise. In Chapter 6, when we presents our design issues, according to the experiments conducted by LEDA [41], we choose binary heap because its implementation in LEDA outperforms the implementation of Fibonacci heap. LEDA is used as the library and platform to implement our algorithms. More details about LEDA is introduced in Chapter 6.

Figure 2.11: (a) A heap viewed as (a) a binary tree, (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number next to a node is the corresponding index in the array.

the tree. This procedure goes through the remaining nodes of the tree and runs *Heapify* on each one. It takes a linear time to build a heap from an unordered set of elements.

3. *Extract-Max* or *Extract-Min*: returns the node with maximum or minimum key which is stored at the root, and calls *heapify* to maintain heap property. It allows the heap data structure to be used as a priority queue. This procedure runs in $O(\log n)$ time since it performs a constant amount of work on top of the $O(\log n)$ time for *Heapify*.
4. *Insert*: inserts a node with a given key value into the heap. To do so, it first expands the heap by adding a new leaf to the tree. Then it traverses a path from this leaf toward the root to find a proper place for the new element. The running time of this procedure is $O(\log n)$ since the path traced from the new leaf to the root has length $O(\log n)$.

2.3.3.2 Binomial Tree, Binomial Heap, and Fibonacci Heap

The *binomial tree* B_k is an ordered tree defined recursively. As shown in Figure 2.12(a), the binomial tree B_0 consists of a single node. The binomial tree $B_k (k > 0)$ consists of two binomial trees B_{k-1} that are *linked* together: the root of one is the leftmost child of the root of the other. Figure 2.12(b) shows the binomial trees B_0 through B_3 . The binomial tree B_k can also be viewed as a tree consisting of the root whose subtrees are $B_{k-1}, B_{k-2}, \dots, B_1$ and B_0 from left to right, shown in Figure 2.12(c). Note that the root has degree k .

Figure 2.12: (a) The recursive definition of the binomial tree B_k . Triangles represent rooted subtrees. (b) The binomial tree B_0 through B_3 . (c) Another way of viewing the binomial tree B_k .

A *binomial heap* H is a set of binomial trees that satisfies the following *binomial-heap properties*:

1. Each binomial tree in H is *heap-ordered*: the key of a node is greater than or

Procedure	binary heap (worst-case)	binomial heap (worst-case)	Fibonacci heap (amortized)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
Extract-Min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
Union	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
Decrease-Key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

Table 2.1: Running times for operations on binary heap, binomial heap, and Fibonacci heap. The number of items in the heap(s) at the time of an operation is denoted by n .

equal to the key of its parent.

2. There is at most one binomial tree in H whose root has a given degree.

The first property says that the root of a heap-ordered tree contains the smallest key in the tree. The second property implies that an n -node binomial heap H consists of at most $\lfloor \log n \rfloor + 1$ binomial trees.

Like a binomial heap, a *Fibonacci heap* is a collection of heap-ordered trees. Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered. The children of a node in a Fibonacci heap are linked together in a circular, doubly linked list, which we call the *child list* of the node. The roots of all the trees in a Fibonacci heap are also linked together into a circular, doubly linked list, which we call the *root list* of the Fibonacci heap. Fibonacci heaps differ from binomial heaps in that they have more relaxed structures, allowing for improved asymptotic time bounds.

Binomial heaps and Fibonacci heaps are known as *mergeable heaps* because they outperform binary heap when two heaps are unioned into a new heap. Table 2.1 shows the comparison of running time of the basic operations of binary, binomial and Fibonacci heaps. Note: *Make-heap* creates and returns a new heap containing no elements. *Union* creates and returns a new heap that contains all the nodes of two

input heaps which are “destroyed” later by this operation. *Decrease-Key* assigns to the input node within the heap the new key value which is assumed to be no greater than its current key value. *Delete* operation deletes the input node from the heap.

In an amortized analysis, the time required to perform a sequence of data structure operations is averaged over all the operations performed. The idea of amortized analysis is that the average cost of an operation may have a better worst case cost than each operation on its own. Amortized analysis differs from average-case analysis in that probability is not involved: an amortized analysis guarantees the average performance of each iteration in the worst case.

From a theoretical point of view, Fibonacci heaps are especially desirable when the number of Extract-Min and Delete operations is small relative to the number of other operations performed. For example, some algorithms for graph problems may call Decrease-Key once per edge. For graphs with many edges, the $O(1)$ amortized time of each call of Decrease-Key adds up to a big improvement over the $O(\log n)$ worst-case time of binary or binomial heaps. From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less attractive than ordinary binary heaps.

2.3.4 Bin Packing

Bin packing arises in a variety of packaging and manufacturing problems. Suppose we are manufacturing widgets with parts cut from sheet metal. To minimize cost and waste, we seek to lay out the parts so as to use as few fixed-size metal sheets as possible. It is also a recurring task in distribution, for instance, scheduling of identical parallel processors so as to minimize the total completion time.

A basic packing problem is known as the one-dimensional bin packing problem. It is to pack a given set of objects having different sizes(or value) into a minimum number of equal-sized bins. The name of the problem, one-dimensional bin packing, is due to the fact that the objects have one dimension such as cost, time, size or weight, etc. that is used for the object’s value. The objective is to minimize the

number of bins. This problem is NP-hard [6, 35].

There are several heuristics to find an approximate solution. The more popular methods are: first-fit, best-fit, first-fit with a sorted list, and best-fit with a sorted list, etc. The first-fit algorithm takes items in the order they come and places them in the first bin in which they can fit. An alternative strategy is that first order the items from largest to smallest, then place them sequentially in the first bin in which they fit.

Summing up, this chapter presents the background on graph separation, including the preliminaries, the key results in planar graph separation, graph separator theorems, and some well-known graph partitioning algorithms with heuristics. Besides, it briefly shows our contributions to planar graph separation. Finally, it reviews the previous results including algorithms and data structures that are employed in our algorithm and implementation.

Chapter 3

Partition Planar Graphs with Costs and Weights

There are many ways to partition planar graphs. For instance, consider a planar graph of m -by- n grid of nodes, where each node is connected to its neighbors to the north, south, east and west. Clearly, the graph can be separated by removing m or n nodes along a column or a row in the “middle” of the grid. Another example is Jordan curve theorem [29]. It says a closed curve divides a plane into exactly two connected regions, the “inside” and the “outside” of the curve. Lipton and Tarjan[38] were the first to formally prove the existence of a $\sqrt{n}$ -separator theorem for all planar graphs. In case of any triangulated planar graph with nonnegative vertex weights, Aleksandrov and Djidjev[2] showed that a vertex separator can be found using their fundamental cycle approach in linear time.

In this thesis, we consider graph $G = (V, E)$ to be an n -vertex connected planar graph with real-valued positive weights $w(v)$ and costs $c(v)$ associated with its vertices. Let t be a real in $(0, 1)$. A set of vertices S is called a t -separator if their removal from G leaves no component of weight exceeding $tw(G)$. The quality of a separator can be measured by its cost. The *cost of a separator* S , denoted by $c(S)$, is the total cost of the vertices in S .

In the rest of this chapter, the outline of our approach is introduced first. Second, the construction of t -separator is provided and the cost of separator is evaluated.

3.1 Outline of Our Approach

Our algorithm is based on the approach introduced in [38] and extended in [2, 9]. In this approach, construction of the separator relies on Dijkstra's single source shortest path (SSSP) tree T rooted at a dummy vertex ρ of zero weight and zero cost that is added to G and connected to at least one of its vertices (so that planarity is preserved). The distance in G is related to the cost of vertices. That is, the cost of traveling along a single edge equals the cost of its target vertex and the cost of a path is the sum of the costs of the edges in that path. In the uniform case, where all vertices have equal costs, T is simply the BFS tree rooted at ρ .

Our algorithm constructs the separator in three phases. In the first phase (or phase I), a set of levels in T are selected and vertices in these levels are added to the separator. Intuitively, a level in T consists of the vertices that are at the “same” distance¹ from ρ and its removal partitions the graph into two disjoint subgraphs induced by the vertices “below” and “above” that level. After the completion of the first phase the removal of the levels in current separator partitions G into components $G_0, \dots, G_p$ with relatively “short” SSSP trees.

In the second phase (or phase II), each connected component G_j whose weight exceeds $tw(G)$ is further partitioned as follows. First, a dummy vertex ρ_j of zero weight and zero cost is added, then the component is triangulated and a SSSP tree T_j consistent with T is constructed. By consistent we mean that the distance between any two vertices in T_j is at most their distance in T . Then G_j is partitioned by means of fundamental cycles with respect to T_j . Recall that a fundamental cycle is a cycle consisting of a single non-tree edge plus the tree path between its endpoints. We find a low cost set of fundamental cycles whose removal partitions G_j into components of

¹If all vertices have the same cost, a level in T consists of the vertices that are at the same distance from ρ . Otherwise, the distances are not the same.

weights at most $tw(G)$ by applying the *separation tree* technique, which is described in details in [2]. Vertices in this set of fundamental cycles are added to the separator. After the second phase we obtain a set of vertices S_t in the separator whose removal leaves no component of weight bigger than $tw(G)$ and whose cost satisfies the following equation.

$$c(S_t) = \sum_{v \in S_t} c(v) \leq 4\sqrt{2\sigma(G)/t} \quad (3.1)$$

where $\sigma(G) = \sum_{v \in V} (c(v))^2$. In terms of satisfying the theoretical bounds, we can stop after the second phase.

In the third phase (or phase III), called *packing*, we “clean up” the separator by removing vertices that are not adjacent to at least two different components as well as by merging neighboring components of small weight into a new component provided that the weight of the new one is still smaller than $\alpha tw(G)$, α is a constant to adjust $tw(G)$.

3.2 Construction of t -separator

We begin this section by a short discussion of the second phase of the algorithm, that is partitioning by means of fundamental cycles.

3.2.1 Partitioning by Fundamental Cycles

Let $G = (V, E)$ be a triangulated planar graph, where each vertex $v \in V$ has weight $w(v)$ and cost $c(v)$. Let T be a spanning tree of G rooted at a vertex ρ . Define the radius of T to be the cost of the longest path in T originating at ρ and denote it by $r(T)$. In [2] it is shown that there exists a set of fundamental cycles $C_1, C_2, \dots, C_q$ with $q \leq \lfloor 2/t \rfloor$ whose removal leaves no component with weight exceeding $tw(G)$. Thereafter, denote the set of vertices in these cycles by C , then C is a t -separator and its cost can be estimated by

$$\begin{aligned}
 c(C) &= \sum_{v \in C} c(v) \leq \sum_{i=1}^q c(C_i) \leq 2qr(T) \\
 &\leq 2\lfloor 2/t \rfloor r(T)
 \end{aligned} \tag{3.2}$$

Here we used the fact that any fundamental cycle in T has cost at most $2r(T)$ [2].

Theorem 1 *Let G be a triangulated planar graph with positive weights and costs on vertices. Let T be a spanning tree of G with radius $r(T)$. There exists a set C of fundamental cycles forming a t -separator such that its cost does not exceed $2\lfloor 2/t \rfloor r(T)$.*

Figure 3.1: Two triangles represent two faces of a graph G . Each face is mapping to a vertex of separation graph $S(G, T)$ of G with respect to spanning tree T rooted at ρ . A tree path from ρ to B and from ρ to C are drawn in bold. A non-tree edge (B, C) of G maps to an edge (S_{ABC}, S_{CBD}) of $S(G, T)$. (B, C) and the tree path to B and to C form a fundamental cycle of G .

Aleksandrov and Djidjev [2] introduce a technique to construct fundamental-cycles by means of computing an edge separator of the separation graph $S(G, T)$ of G with respect to spanning tree T . For example, Figure 3.1 shows two faces $\triangle ABC$ and $\triangle CBD$ of G . Each face maps to a vertex of $S(G, T)$. Edge (B, C) is a non-tree edge of G , and it maps to the edge (S_{ABC}, S_{CBD}) of $S(G, T)$. (B, C) and the tree path to B and C form a fundamental cycle of G . So, an edge in the edge separator of $S(G, T)$ represents a non-tree edge in G , and a non-tree edge of G forms a fundamental-cycle which partitions G . Aleksandrov and Djidjev proved that it takes optimal linear time

to construct such a separator for sparse graph. We employ this technique in the second phase of our algorithm.

Next, we describe the first phase of our algorithm, that partitions G into components with “short” SSSP trees by using levels.

3.2.2 Partitioning by Levels

Assume for simplicity that our graph G is connected and let T be a SSSP tree rooted at ρ . Assume that vertices of G are enumerated by their distance to ρ , i.e.,

$$0 = d(\rho) < d(v_1) \leq \dots \leq d(v_n) = r(T)$$

where n is the number of vertices in G and $d(v_i)$ denotes the distance from ρ to v_i .

Definition 2 Let x be real in $[0, r(T)]$. Denote the set of tree edges of G by $E(T)$. We define a set of edges $E(x)$ by

$$E(x) = \{e = (u, v) : e \in E(T), d(u) < x \leq d(v)\}.$$

Define a set of vertices $L(x)$ called level by

$$L(x) = \{v : e = (u, v) \in E(x), d(u) < d(v)\}.$$

Lemma 1 Let x, y be reals in $[0, r(T)]$. If the interval $[x, y)$ does not contain any of the distances $d(v_i)$, then $L(x) = L(y)$.

Proof: If $x = y$, then the claim is obviously true. Without loss of generality, let $x \leq y$. First, we show that $E(x) = E(y)$. For an edge $e = (u, v) \in E(x)$, by definition we have $d(u) < x \leq d(v)$. Since (x, y) does not contain $d(v_i)$ for $i = 0, \dots, n - 1$, then, $d(u) < y \leq d(v)$ holds. So $e \in E(y)$. On the other hand, let $e = (u, v) \in E(y)$, similarly, we can prove $e \in E(x)$ as well. Therefore, we proved $E(x) = E(y)$. Immediately, $L(x) \equiv L(y)$ follows from $E(x) = E(y)$. $\square$

Corollary 1 *There are at most n different levels in G , i.e., for any $x \in [0, r(T)]$ the level $L(x)$ coincides with one of the levels $L(d(v_1)), \dots, L(d(v_n))$.*

Any of the levels $L(x)$, $x \in (0, r(T))$, cuts G into two subgraphs as stated in the following lemma.

Lemma 2 *If a real $x \in (0, r(T))$, then the vertices of G can be partitioned into three disjoint subsets V_- , $L(x)$ and V_+ defined by $V_- = \{v : d(v) < x\}$, and $V_+ = \{v : d(v) > x, v \notin L(x)\}$. No edge in G joins V_- and V_+ .*

Proof: (By contradiction) Assume there exists a tree edge $e = (u, v)$ such that $u \in V_-$ and $v \in V_+$. So, $d(u) < x$ and $d(v) > x$. By definition, $e \in E(x)$. Therefore $v \in L(x)$ which contradicts that V_+ and $L(x)$ are disjoint vertex sets. Assume there exists a

Figure 3.2: No edge in G joins V_- and V_+ . The bold lines represent the shortest paths $P_T(\rho, u) : \rho \rightsquigarrow u$, and $P_T(\rho, v) : \rho \rightsquigarrow v$, respectively. Edge $e = (u, v)$ is a non-tree edge, and edge $e' = (w, v)$ is a tree edge.

non-tree edge $e = (u, v)$ such that $u \in V_-$, and $v \in V_+$. Denote by $P_T(\rho, u) : \rho \rightsquigarrow u$ and $P_T(\rho, v) : \rho \rightsquigarrow v$ the tree paths from ρ to u and to v , respectively (See Figure 3.2). Consider the path from ρ to v through vertex u , denote by $P(\rho, v) : \rho \rightsquigarrow u \xrightarrow{e} v$. Let

w be the predecessor of v on the path $P_T(\rho, v)$ and $w \notin V_-$. So, $d(u) < x \leq d(w)$. The tree path from ρ to v becomes $P_T(\rho, v) : \rho \rightsquigarrow w \xrightarrow{e'} v$. For the edge $e' = (w, v)$, $c(e') = c(v) = c(e)$. Therefore, we have

$$\begin{aligned} c(P_T(\rho, v)) &= d(w) + c(e') \\ &> d(u) + c(e) \\ &= c(P(\rho, v)) \end{aligned}$$

which contradicts that $P_T(\rho, v)$ is the tree path, i.e. the shortest path from ρ to v . Combining the two cases, we prove the claim. $\square$

Therefore, removal of a level $L(x)$ with $x \in (0, r(T))$ partitions G into two subgraphs G_- and G_+ induced by the sets V_- and V_+ , respectively. The next lemma estimates the integral on $c(L(x))$, which appears in the estimation on the cost of our separators.

Lemma 3 $\int_0^{r(T)} c(L(x))d(x) = \sigma(G) = \sum_{v \in V} (c(v))^2$.

Proof: Let $e = (\text{source}(e), \text{target}(e))$ be a tree edge. Recall that the distance in G is related to the cost of vertices, and the cost of traveling along a single edge equals the cost of its target vertex. So, $d(\text{source}(e)) < d(\text{target}(e))$ holds. $c(e) = c(\text{target}(e)) = d(\text{target}(e)) - d(\text{source}(e))$ holds as well. We define function $\delta_e(x)$ to be 1 if $e \in E(x)$, and 0 otherwise.

$$\begin{aligned} \int_0^{r(T)} c(L(x))d(x) &= \int_0^{r(T)} \left(\sum_{e \in E(x)} c(\text{target}(e)) \right) dx = \int_0^{r(T)} \left(\sum_{e \in E(x)} c(e) \right) dx \\ &= \int_0^{r(T)} \left(\sum_{e \in E(x)} c(e) \cdot 1 + \sum_{e \in E \setminus E(x)} c(e) \cdot 0 \right) dx \\ &= \int_0^{r(T)} \left(\sum_{e \in E} c(e) \cdot \delta_e(x) \right) dx \end{aligned}$$

$$\begin{aligned}
 &= \sum_{e \in E(T)} \left(c(e) \int_0^{r(T)} \delta_e(x) dx \right) \\
 &= \sum_{e \in E(T)} \left(c(e) \int_{d(\text{source}(e))}^{d(\text{target}(e))} \delta_e(x) dx \right) \\
 &= \sum_{e \in E(T)} c(e) (d(\text{target}(e)) - d(\text{source}(e))) \\
 &= \sum_{e \in E(T)} (c(e))^2 \\
 &= \sum_{e \in E(T)} (c(\text{target}(e)))^2 \\
 &= \sum_{v \in V} (c(v))^2 \\
 &= \sigma(G)
 \end{aligned}$$

That completes our proof. $\square$

Lemma 2 and Theorem 1 can be used together for obtaining a number of t -separators in G . Namely, let $i_1, \dots, i_p$ be a collection of increasing indices, i.e. $0 = i_0 < i_1 < \dots < i_p < i_{p+1} = n + 1$. Removal of the levels $L(d(v_{i_1})), \dots, L(d(v_{i_p}))$ partitions G into subgraphs $G_0, G_1, \dots, G_p$. For each of the graphs G_j whose weight exceeds $tw(G)$ we construct a $tw(G)/w(G_j)$ -separator S_j by fundamental cycles with respect to a spanning tree T_j consistent with T . Recall that a tree T_j is called consistent with T if the distances in T_j are at most the distances in T . In case that $w(G_j) \leq tw(G)$ we assume that $S_j = \emptyset$. Separators S_j are constructed as follows:

1. Consider the graph G_0 . We triangulate it and then find an SSSP tree T_0 rooted at ρ . Obviously, T_0 is consistent with T . Applying Theorem 1 we obtain a $tw(G)/w(G_0)$ -separator S_0 such that

$$\begin{aligned}
 c(S_0) &\leq 2 \lfloor 2w(G_0)/tw(G) \rfloor r(T_0) \\
 &\leq 2 \lfloor 2w(G_0)/tw(G) \rfloor (d(v_{i_1-1}) - d(\rho))
 \end{aligned}$$

2. Next, consider the graph G_1 . We connect ρ to all the vertices in G_1 adjacent to $L(d(v_{i_1}))$ in G . The resulting graph is planar. We triangulate that graph

and find an SSSP tree T_1 rooted at ρ . The tree T_1 is consistent with T . Using Theorem 1 we construct a $tw(G)/w(G_1)$ -separator S_1 such that

$$\begin{aligned} c(S_1) &\leq 2 \lfloor 2w(G_1)/tw(G) \rfloor r(T_1) \\ &\leq 2 \lfloor 2w(G_1)/tw(G) \rfloor (d(v_{i_2-1}) - d(v_{i_1})) \end{aligned}$$

3. In this way, for $j = 0, 1, \dots, p$, we obtain a set of vertices S_j that is $tw(G)/w(G_j)$ -separator for G_j and such that

$$c(S_j) \leq 2 \lfloor 2w(G_j)/tw(G) \rfloor (d(v_{i_{j+1}-1}) - d(v_{i_j})) \quad (3.3)$$

Clearly, the union of the levels $L(d(v_{i_1})), \dots, L(d(v_{i_p}))$ and separators $S_0, \dots, S_p$ forms a t -separator for G . We denote that separator by $S(i_1, \dots, i_p)$. Its cost is estimated by

$$\begin{aligned} c(S(i_1, \dots, i_p)) &= c\left(\bigcup_{j=1}^p L(d(v_{i_j}))\right) + c\left(\bigcup_{j=0}^p S_j\right) \\ &\leq c\left(\bigcup_{j=1}^p L(d(v_{i_j}))\right) + \sum_{j=0}^p c(S_j) \\ &\leq c\left(\bigcup_{j=1}^p L(d(v_{i_j}))\right) + 2 \sum_{j=0}^p \left\lfloor \frac{2w(G_j)}{tw(G)} \right\rfloor (d(v_{i_{j+1}-1}) - d(v_{i_j})) \end{aligned}$$

where i_{p+1} is assumed $n+1$. Let $\Gamma(i_1, \dots, i_p)$ be the upper bound of $c(S(i_1, \dots, i_p))$.

$$\begin{aligned} c(S(i_1, \dots, i_p)) &\leq c\left(\bigcup_{j=1}^p L(d(v_{i_j}))\right) + 2 \sum_{j=0}^p \left\lfloor \frac{2w(G_j)}{tw(G)} \right\rfloor (d(v_{i_{j+1}-1}) - d(v_{i_j})) \\ &= \Gamma(i_1, \dots, i_p) \end{aligned} \quad (3.4)$$

How good is such a separator? Selecting different sets of levels may result in different $\Gamma(i_1, i_2, \dots, i_p)$ value. It is desirable to minimize the cost of the separator by finding a set of indices of levels which minimized $\Gamma(i_1, i_2, \dots, i_p)$. In the next theorem we prove that there exists a t -separator $S(i_1, \dots, i_p)$ of small cost by an appropriate choice of the indices $i_1, \dots, i_p$.

Theorem 2 *Let G be an n -vertex planar graph with real-valued positive weights and costs associated to its vertices. For any $t \in (0, 1)$ there exists a t -separator S_t such that $c(S_t) \leq 4\sqrt{2\sigma(G)}/t$ where $\sigma(G) = \sum_{v \in V} (c(v))^2$. Separator S_t can be found in $O(n)$ time in addition to the time for computing an SSSP tree in G .*

Proof: Let $h = \sqrt{t\sigma(G)}/8$. For $j = 0, \dots, p$ define equidistant points $y_j = jh$ in $[0, d(v_n)]$, where $p = \lfloor d(v_n)/h \rfloor$. Points $y_1, \dots, y_p$ divide the interval $[0, d(v_n)]$ into $p + 1$ subintervals such that first p subintervals have length h and the last one has of length less than h . For $j = 1, \dots, p$, let x_j be a point in $[y_{j-1}, y_j]$ where $c(L(x_j))$ is minimum, i.e., we have $c(L(x_j)) \leq c(L(x))$ for $x \in [y_{j-1}, y_j]$. Further, let i_j be the smallest index such that $x_j \leq d(v_{i_j})$. By Lemma 1, we know that $L(x_j) = L(d(v_{i_j}))$.

Consider now the t -separator $S(i_1, \dots, i_p)$ constructed as described above. By (3.4), for the cost of this separator we have

$$c(S(i_1, \dots, i_p)) \leq c\left(\bigcup_{j=1}^p L(d(v_{i_j}))\right) + 2 \sum_{j=0}^p \left\lfloor \frac{2w(G_j)}{tw(G)} \right\rfloor (d(v_{i_{j+1}-1}) - d(v_{i_j})).$$

We estimate the two terms on the right side separately. For the first term we use the definition of the points x_j and Lemma 3

$$\begin{aligned} c\left(\bigcup_{j=1}^p L(d(v_{i_j}))\right) &\leq \sum_{j=1}^p c(L(x_j)) \leq \sum_{j=1}^p \frac{1}{h} \int_{y_{j-1}}^{y_j} c(L(x)) dx \\ &\leq \frac{1}{h} \int_0^{d(v_n)} c(L(x)) dx = \sigma(G)/h. \end{aligned} \quad (3.5)$$

To estimate the second term we observe that $d(v_{i_{j+1}-1}) - d(v_{i_j}) \leq 2h$ for $j = 0, \dots, p$. Thus

$$2 \sum_{j=0}^p \left\lfloor \frac{2w(G_j)}{tw(G)} \right\rfloor (d(v_{i_{j+1}-1}) - d(v_{i_j})) \leq 4h \sum_{j=0}^p \left\lfloor \frac{2w(G_j)}{tw(G)} \right\rfloor \leq 8h/t \quad (3.6)$$

To obtain the theorem we sum (3.5) and (3.6) and substitute h with its value $\sqrt{t\sigma(G)}/8$.

The time bounds can be established easily. Calculating h takes $O(n)$ time, computing equidistant points and finding the corresponding levels take $O(n)$ time, the

algorithm of computing fundamental cycles runs in $O(n)$ time. Therefore, separator S_t can be found in $O(n)$ time in addition to the time for computing an SSSP tree in G . $\square$

3.2.3 Packing

In the third phase, we “clean up” the separator by removing vertices that are not adjacent to at least two different components as well as by merging neighboring components of small weight into a new component. We relax $tw(G)$ to $\alpha tw(G)$ ² so that we can merge as many small components as we need. Basically, Phase III has two steps: first, we identify a vertex v in the separator such that v is adjacent to only one neighboring component. If the weight of the component plus $w(v)$ is less than or equal to $\alpha tw(G)$, then absorb v to this component. We also identify any vertex in the separator so that it is on the boundary shared by two components of G . We create a graph called *super graph* SG of G such that a component of G is represented by a vertex of SG . The weight of the component is the weight of the vertex. An edge e of SG is created if the two components represented by its endpoints share the same boundary. The vertices of the boundary are associated to e . Figure 3.3 shows an example. Second, we pack the *light* components by finding *light* edges of SG . A *light* edge of SG is an edge whose weight plus weights of its endpoints is less than $\alpha tw(G)$. For instance, if an edge $e = (u, v)$ is a light edge of SG , we want to merge the two components represented by vertex u and v into one, the vertices separates this two components are removed from the separator and absorbed by the new component. Figure 3.4 illustrates such an example. We do this packing until no light edges are left *and* the number of vertices of SG is greater than the number of partitions the user asked.

This chapter introduces our algorithm in detail for finding a vertex separator. In addition, it presents our theorem in evaluating the cost of fundamental cycles, and our vertex t -separator theorem.

² α is a constant, chosen from range $[1.0, 2.0]$. In our implementation of phase III, $\alpha = 1.5$.

Figure 3.3: (a) A graph G is separated into 7 pieces after Phase I and Phase II. (b) A super graph SG of G where a component of G is represented by a vertex of SG , and the vertices of the boundary shared by two components of G are associated with an edge of SG . For instance, component 1 of G is represented by vertex 1 of SG , the vertices on the boundary shared by component 1 and 2 are represented by an edge $(1, 2)$ of SG . The total weight of vertices in component 1 is the weight of vertex 1 of SG . The weight of the vertices separating component 1 and 2 is associated with the corresponding edge $(1, 2)$ of SG .

Figure 3.4: (a) Super graph SG has an light edge $(5, 6)$ such that $w(5) + w(6) + w((5, 6)) \leq \alpha tw(G)$ where α is a constant to adjust $tw(G)$. When merging component 5 and 6, edge $(3, 6)$ is deleted, the vertices of G and the weight associated with $(3, 6)$ is added to that of edge $(3, 5)$; edge $(4, 6)$ is deleted and a new edge $(4, 5)$ is created, the vertices and the weight associated with $(4, 6)$ is now associated with $(4, 5)$; edge $(6, 7)$ is deleted and new edge $(5, 7)$ is created, the information of $(6, 7)$ is associated with $(5, 7)$. (b) Resulting super graph SG after merging vertex 5 and 6 into vertex $5'$ and deleting the light edge $(5, 6)$. (c) The resulting separated graph G after merging two components 5 and 6 into one.

Chapter 4

Reducing the Cost of Vertex t -Separator

In this chapter we describe a technique that optimizes the choice of levels to be inserted in the separator during the first phase of the algorithm. Recall that in the first phase we select a collection of indices $i_1, \dots, i_p$ and then cut the graph by removing levels $L(d(v_{i_1})), \dots, L(d(v_{i_p}))$. We have shown that the cost of the separator $S(i_1, \dots, i_p)$ resulting after the choice of levels corresponding to these indices is bounded by $\Gamma(i_1, \dots, i_p)$ from Inequality 3.4. To prove our main result (Theorem 2) in Section 3.2.2, we have shown how to choose a collection of indices such that the value of Γ does not exceed $4\sqrt{2\sigma(G)/t}$. As we have seen, these indices can be found efficiently in linear time and the cost of the resulting separators $\Gamma(i_1, \dots, i_p)$ is asymptotically optimal. Hence, such a choice is satisfactory from an asymptotic complexity point of view.

In practice however we may apply a simple and natural heuristic. That is, we may find a collection of indices for which the value of Γ is minimum and then use the corresponding levels to partition the graph. It is not possible to argue that such a choice is *superior* in all cases since we are minimizing a function that is an upper bound on the cost of the separator but not the cost itself. Nevertheless, our experiments

(see Table 6.8 and Figure 6.11) show that this heuristic significantly improves the cost of the separators and is computational efficient. In the remaining of this section we describe an algorithm for finding a collection of indices $i_1, \dots, i_p$ such that the value $\Gamma(i_1, \dots, i_p)$ is minimum.

4.1 Reformulating the Problem as a Shortest Path Problem

Notice that the number of different levels is not always n . If two vertices have the same distance to the root ρ then their levels coincide. Assume that there are $n' \leq n$ vertices with distinct distances from ρ . Assume that vertices with the same distance are grouped and let for simplicity denote these groups with $v_1, \dots, v_{n'}$ so that $0 = d(\rho) < d(v_1) < \dots < d(v_{n'}) = r(T)$. Thus we have n' distinct levels $L(d(v_i))$, $i = 1, \dots, n'$ to choose from.

Consider graph $H = (V(H), E(H))$ is defined by

$$V(H) = \{0, 1, \dots, n' + 1\} \quad (4.1)$$

$$E(H) = \{(i, j) : 0 \leq i < j \leq n' + 1\} \quad (4.2)$$

See Figure 4.1. A vertex of H , except vertex 0 and $n' + 1$, represents a group of vertices with the same distance from ρ in G . It also represents a level in G . Vertex 0 and $n' + 1$ are dummy ones, representing no vertex of G . The cost (or weight) of a vertex of H is the cost (or weight) of the level it represents. We call H as a *level graph* and assign costs to its edges so that the cost of path from 0 to $n' + 1$ and passing through $i_1, \dots, i_p$ is exactly $\Gamma(i_1, \dots, i_p)$. This is achieved by defining the cost of an edge (i, j) as

$$\text{cost}(i, j) = c(L(v_j) \setminus L(v_i)) + 2[2w(G_{i,j})/tw(G)](d(v_{j-1}) - d(v_i)) \quad (4.3)$$

where $L(v_0) = L(v_{n'+1}) = \emptyset$ and $w(G_{i,j})$ is the total weight of the vertices with distances between $d(v_i)$ and $d(v_j)$ and not in $L(d(v_i))$. Now the problem of finding a

Figure 4.1: Illustration of levels graph H .

collection of indices minimizing Γ can be solved by finding the shortest path between vertices 0 and $n' + 1$ in level graph H .

4.2 Efficient Computation of Shortest Paths in H

Dijkstra's shortest path algorithm can be used to find the shortest path between 0 and $n' + 1$, and hence a collection of indices for which T is minimized. The problem with directly applying Dijkstra's algorithm on H is that it takes $\theta((n')^2)$ time, which can be $\Omega((n')^2)$. Below we discuss how the special structure of graph H can be exploited so that Dijkstra's algorithm will not need to explore all the edges of H .

The classical Dijkstra's algorithm maintains a priority queue Q storing the vertices of the graph, where the key of a vertex i in Q is the shortest distance from 0 to i found so far. In each iteration the algorithm extracts the vertex with minimum key from Q , which we refer to as *current vertex* and then the edges incident to the current vertex are *relaxed*¹.

Notice that we can not precompute the cost of the edges of H since this will take $\Omega((n')^2)$. Next, we describe briefly the main point in our implementation of Dijkstra's algorithm on H , we precompute the cost and weights of the sets of vertices $L(d(v_{i+1})) \setminus L(d(v_i))$ for $i = 1, \dots, n'$ in $O(n)$ time. Using these precomputed values, we compute the cost of an edge (i, j) in $O(1)$ time provided that we know the cost of

¹Relaxed means the distances to the vertices adjacent to the current vertex are updated and operation decrease key in Q is performed when shorter distance is found.

$(i, j - 1)$ or $(i, j + 1)$. This leads to the following set of rules for our implementation.

Rule 1 *Edges incident to the current vertex are relaxed following their consecutive order (increasing or decreasing).*

For a vertex $i \in V(H)$ denote by $dist(i)$ the cost of the shortest path from 0 to i . From Equation 4.3 we notice that the cost of an edge (i, j) in H is a sum of two terms. We call them *cost part* and *weight part* and denote them by $cost_c(i, j)$ and $cost_w(i, j)$. We observe that the weight part of an edge is an increasing function with respect to the target vertex. Hence, we have the next rule.

Proposition 1 *If $i < j$ and $dist(i) \geq dist(j)$ then there is a shortest path from 0 to $n' + 1$ that does not include i .*

Proof: Let k be any vertex after j . We want to show that i can not be on the shortest path from 0 to $n' + 1$ because the path through j to k is shorter than the path through i to k . Figure 4.2 illustrate such situation.

$$\begin{aligned}
 & dist(j) + cost(j, k) - (dist(i) + cost(i, k)) \\
 &= dist(j) + cost_c(j, k) + cost_w(j, k) - (dist(i) + cost_c(i, k) + cost_w(i, k)) \\
 &= dist(j) + cost_w(j, k) - (dist(i) + cost_w(i, k)) \\
 &\leq 0
 \end{aligned}$$

So, i can not be on the shortest path from 0 to $n' + 1$. We use the fact that $cost_c(j, k) = cost_c(i, k)$ because it is the cost of the level that k represents. $\square$

Rule 2 *The indices of the consecutive current vertices must increase.*

That means if $\text{Extract_min}(Q)$ produces a vertex with smaller index than the previous current vertex, no relaxation is done. Furthermore, from Theorem 2 it follows that $dist(n' + 1) \leq 4\sqrt{2\sigma(G)/t}$. Therefore we do not need to consider paths longer than $4\sqrt{2\sigma(G)/t}$ during the implementation of Dijkstra's algorithm.

Figure 4.2: Illustration of Proposition 1. k is any vertex after j and i is before j . $dist(j)$ represents the shortest distance from 0 to j known so far. Since the path through j to k is shorter than the path through i to k , i can not be on the shortest path from 0 to $n' + 1$.

Rule 3 If i is the current vertex and $dist(i) + cost_w(i, j) > 4\sqrt{2\sigma(G)}/t$ then edges (i, j') with $j' \geq j$ are not relaxed.

Applying this rule during the relaxation we can discard all edges “after” some edge. We define a vertex $jump(i)$ to be the smallest index such that $w(G_{i, jump(i)}) > tw(G)$. At some point, say vertex k where $0 \leq k \leq n' + 1$, $w(G_{k, jump(k)})$ will not exceed $tw(G)$ simply because there are not enough vertices after k . For such a vertex k , $jump(k)$ is vertex $n' + 1$. See Figure 4.3.

Figure 4.3: For each vertex i of graph H where $0 \leq i \leq n' + 1$, there is a vertex $jump(i)$ computed. An edge here connects vertex i and $jump(i)$.

Next proposition shows that we can discard all edges “before” some edge under certain conditions.

Proposition 2 If $i' < i < j$ and $dist(i') + cost(i', j) \leq dist(i) + cost(i, j)$ then $dist(i') + cost(i', j') \leq dist(i) + cost(i, j')$ for any $i < j' < j < jump(i)$.

Proof: Figure 4.4 shows the situation of Proposition 2. We know the following holds

$$dist(i) - dist(i') \geq cost(i', j) - cost(i, j) \geq cost_w(i', j) - cost_w(i, j)$$

We prove the claim by showing

$$\text{dist}(i) - \text{dist}(i') \geq \text{cost}(i', j') - \text{cost}(i, j') \geq \text{cost}_w(i', j') - \text{cost}_w(i, j')$$

and

$$\text{cost}_w(i', j) - \text{cost}_w(i, j) \geq \text{cost}_w(i', j') - \text{cost}_w(i, j').$$

By definition of $\text{jump}(i)$, $\text{cost}_w(i, j) = \text{cost}_w(i, j') = 0$ holds. Also, $\text{cost}_w(i', j) \geq$

Figure 4.4: Illustration of the situation of Proposition 2

$\text{cost}_w(i', j')$ holds because this weight part is an increasing function with respect to the target vertex. Hence, we prove this proposition. $\square$

Rule 4 *If i is the current vertex and the relaxation of an edge (i, j) does not result in smaller distance to j then edges (i, j') with $j' < j$ are not relaxed.*

The above four rules suggest the following implementation of Dijkstra's algorithm on H . For each vertex i we compute a vertex $\text{jump}(i)$. Assume that i is a current vertex whose edges are to be relaxed. We relax the edge $(i, \text{jump}(j))$ first. Then edges before $\text{jump}(i)$ are relaxed in decreasing order until the first unsuccessful relaxation is encountered (Rule 4). Next, edges after $(i, \text{jump}(i))$ are relaxed in their increasing order until Rule 3 applies. Figure 4.5 shows the set of edges relaxed by any current vertex i .

Our experiments show that this results in a very efficient procedure and timings suggest that the running time is at most $O(n')$ in case of positive integer costs on vertices. So far, we do not have a rigorous proof of this bound, whereas some of the special cases are characterized in the following lemma.

Figure 4.5: The set of edges relaxed by any current vertex i .

Lemma 4 *In addition to the time for computing SSSP tree of G in the preprocessing stage, if the costs of vertices are positive integers then a collection of indices minimizing Γ can be found in $O(\min(n^2, \sigma(G)/t))$ time. In the uniform case, when all vertices have the same cost, the time bound is $O(\min(n^2, n/t))$.*

Proof: We consider the case of integer costs. The work done by the algorithm can be separated into three parts. First preprocessing, where $\text{jump}(i)$ vertices and costs $\text{cost}(i, \text{jump}(i))$ are computed in $O(n)$. Second, work to relax edges. And finally work for Extract_min and Relaxation(Decrease_key) operations in Q .

Consider now the work done by the algorithm to relax edges. Since relaxation in our case takes $O(1)$ work (Decrease_key time not counted here) we will estimate the number of edges considered for relaxation.

First we argue that the total number of the edges before jump-vertices is $O(n)$. This follows from the observation that for any vertex j in H at most one such edge ending at j can be successfully relaxed (Proposition 2).

So far, we did not use that our costs are integer. This comes into play in the next two estimates. To estimate the number of after jump-vertex edges let us estimate how many such edges can be relaxed from a fixed current vertex i . The weight part of any such edge (i, j) must be at most the upper bound B where $B = 4\sqrt{2\sigma(G)/t}$ (Rule 3), i.e. $\text{cost}_w(i, j) = 2\lfloor 2w(G_{i,j})/tw(G) \rfloor (d(v_{j-1}) - d(v_i)) \leq B$. Now notice that in case of integer cost $j - i + 1 \leq d(v_{j-1}) - d(v_i)$ and that for after jump-vertex edges $\lfloor 2w(G_{i,j})/tw(G) \rfloor \geq 2$. These inequalities imply $j - i < B/2$, which means that at most $B/2$ after jump-vertex edges are relaxed from a fixed current vertex.

Next we observe that according to Rule 1 we can have at most B current vertices

from which we relax edges since their distances increases by at least 1 in each iteration and the algorithm will find the shortest path to $n' + 1$ in at most B iterations. Therefore the total number of the after-jump edges relaxed by the algorithms is at most $B^2/2$.

Finally consider the cost of `Extract_min` and `Decrease_key` operations performed in the priority queue. If we use Fibonacci heaps we will require $O(\log n)$ time per `Extract_min` and $O(1)$ amortized time per `Decrease_key` operation. This leads to $O(n \log n + B^2)$ time upper bound on the running time on this part of the algorithm. But now we can use the fact that the keys are integers and bounded, we implement the priority queue using an array, which will lead to $O(B^2)$ total time spent for priority queue operations.

We implement a priority queue by employing an array Q of size B with its elements initialized to *nil* and $Q[0] = 0$. Assume now that the algorithm finds a new distance d to a vertex j of H . Then the algorithm checks the contents $Q[d]$ and if this value is less than j then updates $Q[d] = j$. This is the `Decrease_key` operation obeying Rule 1. To implement the `Extract_min` operation we traverse array Q from the left to the right and return the contents of non-*nil* elements of Q . This implementation of the priority queue takes B time to traverse the array (`Extract_min` operations) and $O(1)$ time per `Decrease_key` operation, which totals to $O(B^2) = O(\sigma(G)/t)$.

Summing up, our algorithm of finding a collection of indices minimizing Γ runs in $O(\min(n^2, \sigma(G)/t))$ time for positive integer costs on vertices.² When all vertices have the same cost, the time bound is

$$\begin{aligned} O(\min(n^2, \sigma(G)/t)) &= O(\min(n^2, (\sum_{v \in V} (c(v))^2)/t)) \\ &= O(\min(n^2, n(c(v))^2/t)) = O(\min(n^2, n/t)). \square \end{aligned}$$

In conclusion, this chapter describes our cost-reducing technique that optimizes the choice of levels during the first phase of the algorithm. It illustrates how to

²The purpose of this tuning technique is to consider the edges that will be relaxed in computing SSSP of H . In case of $\sigma(G)/t > n^2$, there are at most n^2 number of edges of H to be relaxed. Therefore, the time complexity is $O(\min(n^2, \sigma(G)/t))$.

reformulate the problem of finding optimal levels as a shortest path problem in graph H . More importantly, this chapter presents our heuristic rules to simplify constructing graph H . The time complexity of the cost-reducing technique is analyzed in case of positive integer costs on vertices and unit costs on vertices.

Chapter 5

Applications

Separators for graphs with vertex weights and costs have applications in efficiently solving combinatorial problems such graph pebbling, and construction of a minimum routing tree, etc. See [9] for further discussion of these and other problems. In this chapter, first, we show how to obtain some existing vertex separator results from Theorem 2. Second, we show how Theorem 2 can be applied to construct an edge t -separator in a weighted planar graph, our edge t -separator theorem, and the other existing edge separator results that our theorem implies.

5.1 Variations of Vertex Separator Theorem

Our vertex separator theorem (Theorem 2) in essence captures many of the existing results on separators in planar graphs. In this section, we show how to obtain a variety of existing vertex separator results, for instance the results of Lipton and Tarjan [38], and Djidjev [9] from Theorem 2, by appropriately setting the costs and weights of vertices in the given planar graph.

Recall that the main theorem of Lipton and Tarjan [38] says: *Let G be any n -vertex planar graph having nonnegative vertex weights summing to no more than one. Then the vertices of G can be partitioned into three sets A , B , C such that no edge*

joins a vertex in A with a vertex in B , neither A nor B has total weight exceeding $2/3$, and C contains no more than $2\sqrt{2}\sqrt{n}$ vertices.

Lipton and Tarjan [38]'s main theorem considers nonnegative vertex weight. Theorem 2 considers real-valued positive vertex costs and weights. In case of uniform cost on vertices, substituting $2/3$ for t , we derive the cost of separator of Theorem 2: $4\sqrt{2\sigma(G)/t} = 4\sqrt{2n/t} = 4\sqrt{3}\sqrt{n}$. The cost of separator Theorem 2 predicts differs from the result in [38] by a small constant $\sqrt{6}$.

Recall that the main theorem of Djidjev [9] says as follows: *Let G be a planar graph with non-negative vertex weights that sum up to one and non-negative vertex costs whose squares sum up to $N(G)$. There exists a $2/3$ vertex separator of G with total vertex cost no more than $\sqrt{8N(G)}$. Such a separator can be found in $O(|G|)$ time.*

Notation $N(G)$ used by Djidjev is the $\sigma(G)$ in our case. Again, we substitute $2/3$ for t to derive the cost of separator of Theorem 2: $4\sqrt{2\sigma(G)/t} = 4\sqrt{3N(G)}$. What Theorem 2 predicts about the cost of separator differs from the result in [9] by a small constant $\sqrt{6}$.

5.2 Application: Construction of Edge t -separator

In this section, we will introduce an application of Theorem 2 in the construction of edge t -separator. First, we show how we form an edge separator from the vertex separator. Recall that a set of edges is called a *edge t -separator* of G , if their removal leaves no component of weight greater than $tw(G)$. Second, we show our main theorem for edge t -separator. At last, we show how to obtain variety of existing edge separator results from this edge t -separator theorem.

Construction of an edge t -separator is one more step from an existing vertex t -separator. The simplest way to do it is for each vertex in the vertex separator, put all its adjacent edges into the edge separator. Each vertex in the vertex separator forms its own region. Figure 5.1(b) shows an example. In the algorithm introduced

next, we apply *best-fit* then *first-fit* strategy of the bin-packing problem to pack the vertices of the vertex separator to their neighboring components as much as possible to eliminate the chance that they form their own regions. This minimizes the number of edges added to the edge separator.

Algorithm: Initially, edge separator set ES is empty. For any vertex v in vertex separator of G , first, find the component c_i adjacent to v with the maximum number of edges. Second, if the weight of this component plus the weight of v is less than $\alpha tw(G)$ ¹, then v is absorbed into the component c_i . Add the other edges incident to v to the set ES . Otherwise, try any other component c_j adjacent to v , if the weight of c_j and the weight of v is less than $\alpha tw(G)$, then v is absorbed to c_j . If none of the adjacent components can absorb v under the weight condition $\alpha tw(G)$, then v by itself forms a new region. All edges incident to v will be added to ES . That is certainly the worst case. In our implementation of this stage, we relaxed $tw(G)$ to $2tw(G)$. Even though this relaxation does not guarantee that the worst case never happens, the experimental results show that this strategy works quite well for most of the cases we encounter. The final set ES is the edge t -separator constructed.

Figure 5.1(a) shows an example: a vertex drawn in big is in the vertex separator, and labeled 0. The left hand side component is drawn partially and labeled 1, and the right hand side component is drawn partially too, and labeled 2. Figure 5.1(b) shows the naive algorithm to construct an edge separator. Edges in the edge separator are drawn in bold. The vertices originally in the vertex separator now form their own regions and their labels are 3, 4, ..., 9. Figure 5.1(c) shows using our algorithm, the vertices in the vertex separator are absorbed into the neighbor components, and their labels represent which component they belong to. This simple example illustrate that our algorithm optimizes the size of edge separator to some point.

Our main theorem of edge t -separator for G presented next guarantees the upper bound of the size of edge separator produced.

Theorem 3 *Let G be an n -vertex planar graph with positive weights on vertices. Let t*

¹ α is a constant in range $[1.0, 2.0]$. In our implementation of this stage, $\alpha = 2.0$

Figure 5.1: Construct an edge separator from a vertex separator. (a) partially shown a graph is partitioned into 2 pieces. The vertices drawn in big are in the vertex separator, and labeled 0. The other vertices drawn in small are labeled 1 or 2, representing the components they belong to. (b) a naive algorithm to construct an edge separator from the vertex separator. Edges in the edge separator are drawn in bold. The vertices originally in the vertex separator now form their own regions and their labels are 3, 4, ..., 9. (c) using our algorithm, the vertices in the vertex separator are absorbed into neighbor components.

be a real in $(0, 1)$ and no vertex of G has weight exceeding $tw(G)$. There exists an edge t -separator consisting of at most $4\sqrt{2\Delta(G)/t}$ edges, where $\Delta(G) = \sum_{v \in V} (deg(v))^2$ and $deg(v)$ is the degree of v . Such separator can be found in $O(n)$ time in addition to computing SSSP tree in G .

Proof: First we set the cost of each vertex as its degree. Then we apply Theorem 2 and obtain a vertex t -separator VS_t whose cost is at most $4\sqrt{2\sum_{v \in V} deg^2(v)/t}$. Since the cost of a vertex is its degree, there are at most $4\sqrt{2\sum_{v \in V} deg^2(v)/t}$ edges incident to vertices in VS_t . Next, for each vertex $v \in VS_t$ we consider components of $G \setminus VS_t$ adjacent to v and if there is a component whose weight plus the weight of v is less than $tw(G)$ (or $\alpha tw(G)$ in our algorithm) we pack v to that component. If there is no such component we make v to constitute its own component. Finally, we form an edge t -separator ES_t by inserting in it all edges incident to vertices in VS_t that join different components. Clearly, ES_t satisfies the requirements of the theorem.

The algorithm for finding a vertex separator introduced in Chapter 3 runs in $O(n)$ time in addition to the time for computing SSSP tree of G in the preprocessing stage. The algorithm for constructing edge separator from vertex separator runs in $O(dn_t)$ time where n_t is the size of the vertex separator produced, and d is the maximum number of neighboring components adjacent to a vertex in vertex separator. n_t is far less than n , d can be treated as a constant comparing to n . So, the time complexity of finding an edge separator of G is $O(n)$ in addition to computing SSSP tree. $\square$

5.3 Variations of Edge Separator Theorem

Our edge separator theorem implies many of the existing edge-separator results. For instance, in [7] it has been shown that any n -vertex planar graph of maximum vertex degree d has a $(2/3)$ -edge separator of cardinality $O(\sqrt{dn})$. Notice that in planar graphs $\sum_{v \in V} (deg(v))^2 \leq 6dn$, where d is the maximum degree in G . Substitute $2/3$ for t of Theorem 3, we have

$$4 \sqrt{2 \sum_{v \in V} (deg(v))^2 / t} \leq 4 \sqrt{12dn/t} = O(\sqrt{dn})$$

So, Theorem 3 implies the edge separator theorem in [7]

Other versions of $(2/3)$ -edge separator, where edges have cost, have been highlighted in Djidjev [9]. It says: *Let G be a planar graph of maximum vertex degree d , with non-negative vertex weights that do not exceed $2/3$ and sum up to one, and non-negative edge costs whose squares sum up to $N(G)$. There exists a $2/3$ edge separator of G of total cost not exceeding $4\sqrt{dN(G)}$, which can be found in $O(|G|)$ time.* Let a be the maximum edge cost. The result of Djidjev's edge separator becomes

$$4\sqrt{dN(G)} < 4a\sqrt{dn} = O(\sqrt{dn})$$

Therefore, Theorem 3 implies Djidjev's edge separator theorem too.

Briefly, this chapter introduces our edge t -separator theorem, our algorithm for constructing such an edge separator from a vertex separator. It shows how our vertex and edge t -separator theorems imply many existing results in planar graph separation.

Chapter 6

Implementation and Experimental Results

The focus of this chapter is to provide relevant details on the implementation of vertex and edge t -separators and to present our experimental results. Section 6.1 introduces the major design decisions we made for the implementation. Section 6.2 describes the motivation of our experiments, the experimental environment, and the experiments we have conducted. This chapter ends with some experimental results as well as our observation and analysis.

6.1 Design Issue

In spite of the design of our algorithms for vertex and edge t -separators introduced in Chapter 3, 4, and 5, there are other design issues worth mentioning. In this section, we show the design decisions we have made for the implementation of our algorithms. First, we introduce *LEDA* (Library of Efficient Data Types and Algorithms)[41] and why we choose LEDA as a library of data types and algorithms. Second, we present two versions of priority queue implementation for Dijkstra's SSSP: one version is for computing SSSP tree at the preprocessing stage, the other is for optimizing the choice

of levels so that the cost of separator is reduced. This problem is reformulated as a shortest path problem. At last, we show how to apply the strategy of the bin packing problem to obtain an edge separator from a vertex separator.

6.1.1 Why Choose LEDA

The *LEDA* project was started in 1989 and leaded by Mehlhorn and Näher [41]. It is not only a library of data types and algorithms of combinatorial and geometric computing but also a platform on which applications can be built. After more than a decade of improvements, LEDA provides a sizable collection of data types and algorithms in a form that allows them to be used by non-experts. This collection includes most of the data types and algorithms described in textbooks: stacks, queues, lists, sets, dictionaries, ordered sequences, partitions, and priority queues; directed, undirected, and planar graphs, lines, points, and planes; and many algorithms in graph and network theory and computational geometry. LEDA gives a precise and readable specification for each of the data types and algorithms just mentioned. Many data types in LEDA are parameterized, so a user has the flexibility of choosing a data type such as integer, float, double, string, and user defined types, etc. of his/her interest.

In addition, LEDA contains a comfortable data type graph. It offers the standard iterations, addition and deletion of nodes and edges, and arrays and matrices indexed by nodes and edges. This is very convenient when costs and weights are associated with nodes(vertices) or edges of graphs. LEDA contains the most efficient realization know for its types. It is realized in C++ and all its data types and algorithms are stored in the library as precompiled object-modules. This leads to short compile time. Also, LEDA offers an interface to the X11 window system to allow graphical output and mouse input. This feature allows us to visualize our graphs and separators on the screen or to a file easily. For the elegance and ease of use in LEDA, we choose LEDA as a library and a platform to build our implementation of graph separators.

6.1.2 Priority Queue Implementation for Dijkstra's SSSP

The main problem we are facing in this research is to construct a vertex t -separator by finding a set of indices of levels whose removal separates G into components with weight less than $tw(G)$. In case there is a component whose weight exceeds $tw(G)$, Aleksandrov and Djidjev's fundamental-cycle approach [2] is employed to further partition it. In order to reduce the cost of vertex t -separator, this problem is reformulated as a variation of the shortest path problem (see Chapter 4). Efficient computation of such shortest paths is the focus of our design.

Dijkstra's SSSP algorithm maintains a priority queue Q storing the vertices of the graph, where the key of a vertex i in Q is the shortest distance from 0 to i found so far. If costs on vertices are positive real, then the key of a vertex is a real too. Normally, a priority queue implemented by binary or Fibonacci heap will do the job. If costs on vertices are positive integers, then the key is integer too. For the latter case, we design a special implementation of priority queue so that the running time of computing such shortest path is $O(\sigma(G)/t)$. In the uniform case, when all vertices have the same cost, the time bound is $O(n/t)$ (see the proof of Lemma 4 in Section 4.2).

As mentioned briefly in the proof of Lemma 4 in Section 4.2, we implement such a priority queue by employing an array Q of size $B + 1$ where $B = 4\sqrt{2\sigma(G)/t}$, with its elements initialized to *nil* and $Q[0] = 0$. Q serves as scoreboard. Index of Q is from 0 to B , representing all possible distance up to B to vertex 0. For an index i , content $Q[i]$ stores the vertex whose distance is i . Let $dist[j]$ denote the distance from vertex j to vertex 0. We explain how the two operations *Extract_min* and *Decrease_key* are realized for such a priority queue.

1. *Decrease_key*: if our algorithm finds a less distance d for a vertex j , then it updates $dist[j] \leftarrow d$, and checks the contents $Q[d]$: if $Q[d]$ is less than j (meaning if the vertex stored at $Q[d]$ is before j) then updates $Q[d] \leftarrow j$.
2. *Extract_min*: it is implemented by traversing Q from 0 to B . It returns the

content of first non-nil element of Q , which is the vertex with the current shortest distance.

So, this implementation of the priority queue takes B time to traverse the array (Extract_min operations) and $O(1)$ time per Decrease_key operation, which sums to $O(B^2) = O(\sigma(G)/t)$ as the upper bound on the running time for phase I of the algorithm. If we used Fibonacci heaps to implement the priority queue, it would require $O(\log n)$ time per Extract_min and $O(1)$ amortized time per Decrease_key operation. That leads to $O(n \log n + B^2)$ time upper bound on the running time for phase I of the algorithm.

In the preprocessing stage of our algorithm, Dijkstra's SSSP tree is built to compute the distance for each vertex of G . This time, Dijkstra's SSSP is implemented as described in subsection 2.3.2 in Chapter 2. LEDA provides many implementations of priority queues, for instance, Fibonacci heaps, binary heaps, pairing heaps, k -ary heaps, etc. LEDA did tests on the running time of Dijkstra's algorithm with different priority queue implementations, the experimental results say that the implementation with binary heaps outperforms Fibonacci heaps in most of the cases except when testing graphs have 200000 vertices and 500000 edges and the edge weights from 0 to 100000. For this reason, the priority queue implemented in binary heap is chosen instead of Fibonacci heap.

6.1.3 Edge t -separator and Bin Packing

As an application of our vertex separator theorem, an edge separator is obtained from existing vertex separator using the algorithm introduced in Section 5.2. Packing the vertices in vertex separator back to their neighbor components is just like the bin packing problem (see Subsection 2.3.4). In our algorithm, we first use the "best fit" strategy to absorb such a vertex v to its "best fit" neighbor region: the region adjacent to v with the maximum number of edges. This reduces the size of the edge separator. If the "best fit" region can not absorb v because the sum of their weights exceeds $\alpha tw(G)$ (known as the weight condition), then we follow "first fit" strategy, letting

the “first fit” component absorb the vertex. The worst case may happen when none of the adjacent components can absorb v under the weight condition, and v by itself forms a new region. We relaxed the weight condition to at most $2tw(G)(\alpha = 2.0)$ in this algorithm. This relaxation does not guarantee that the worst case never happens, but it reduces the possibility of its occurrence. This modification seems to work well for most of the tests we performed. This algorithm not only takes care of reducing the size of edge separator but also is simple and easy to implement.

6.2 Experiments

This section starts with the goal of our experiments and our system environment, followed by what we implemented and the input parameters of our programs. Then, we present what we tested, experimental results, our observations and analysis. At last, we compare our results with the results shown in [16]

6.2.1 Goal of Experiments and System Environment

The first goal of our experiments is to test our implementations on a large amount of planar graphs of different types against what we predict in our vertex separator theorem and edge separator theorem. Denote the resulting vertex separator by S_t , and edge separator by ES_t . In particular, we collect the following experimental results:

1. the cost of separator $c(S_t)$ and $|ES_t|$
2. the ratio of the cost of separator $c(S_t)$ and $\sqrt{\sigma(G)}/t$ for vertex separator, and the ratio of $|ES_t|$ and $\sqrt{\Delta(G)}/t$ for edge separator¹(in theory, the ratio is $4\sqrt{2}$)
3. the number of components generated
4. the largest and smallest components in terms of weights when cost is one

¹As defined before, $\sigma(G) = \sum_{v \in V} (c(v))^2$, $\Delta(G) = \sum_{v \in V} (deg(v))^2$ and $deg(v)$ is the degree of v .

5. the CPU time of Phase I and II, and Phase III of our algorithms
6. balance, the ratio of maximum and minimum weight of resulting components

The second goal is to test the efficiency of the cost reducing technique described in Chapter 4.

The system environment of our experiments is Sun Ultra 10 Microsystems with 440 MHz clock and 512 Mb memory running Solaris 8 operating system. Our algorithms are implemented in C++ using LEDA version 4.1. The C++ compiler is GNU g++ version 2.95.2.

6.2.2 What We Implemented and Input Parameters

We have implemented the algorithm for constructing vertex t -separator introduced in Chapter 3, and Chapter 4. We have two different implementations of Phase I of the vertex separator algorithm. In the first version we implemented the separator by choosing the levels as described in the proof of Theorem 2. We call this *equi-distant levels partition*. In the second version, we implemented the algorithm using the cost reducing technique presented in Chapter 4. As an application, we have implemented the algorithm for constructing edge separator as described in Chapter 5.

The input of our programs consists of three parameters: input graph name, number of parts into which the input graph is to be partitioned, and an integer representing what costs and weights are associated to vertices of input graphs. For input graphs, we use well known planar graphs available on the web², including finite element meshes (FEM) in 2-dimension (2D), irregular planar graphs, and triangular irregular networks (TIN). Our programs accept input graphs in the LEDA graph format. So we implemented two converters: one is to convert TIN graphs to the LEDA format, the other is to convert the FEM graphs to the LEDA format. Table 6.1 lists a sample of planar graphs used in our experiments.

²e.g., see <http://www.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/graphs.html>

Graph	n	m	Type
trapezoid	819	2286	irregular planar
tapir	1024	2846	irregular planar
airfoil	4253	12289	irregular planar
cycle30k	30001	30001	irregular planar
africa	5000	14798	TIN
australia	5000	14943	TIN
brazil	5000	14777	TIN
deberg	29	68	TIN
europa	5000	14816	TIN
greatlakes	5000	14873	TIN
greenland	5000	14689	TIN
japan	1500	4353	TIN
madagascar	5000	14838	TIN
sanbern	5000	14770	TIN
america	5000	14787	TIN
tin50000	25440	75683	TIN
tin100000	50624	150973	TIN
3elt	4720	13722	FEM (2D)
3elt_dual	9000	13278	FEM (2D)
whitaker	9800	28989	FEM (2D)
whitaker_dual	19190	28581	FEM (2D)
big	15606	45878	FEM (2D)
big_dual	30269	44929	FEM (2D)
crack	10240	30380	FEM (2D)
crack_dual	20141	30043	FEM (2D)
grid1	252	476	FEM (2D)
grid1_dual	224	420	FEM (2D)
netz4504	1961	2578	FEM (2D)
netz4504_dual	615	1171	FEM (2D)
ukerbel	5981	7852	FEM (2D)
ukerbel_dual	1866	3538	FEM (2D)

Table 6.1: Sample input graphs with n vertices and m edges.

program	combinations of cost and weight	results	results
vertex t -separator	costs equal to 1 weights equal to 1.0	Table 6.3	Figure 6.1 Figure 6.2
	costs equal to 1, and weights are random real in range (1.0, 100.0)	Table 6.4	Figure 6.3 Figure 6.4
	costs are random integer in range (1, 100) weights are random real in range (1.0, 100.0)	Table 6.5	Figure 6.5 Figure 6.6
edge t -separator	costs are degrees of vertices weights equal to 1.0	Table 6.6	Figure 6.7 Figure 6.8
	costs are degrees of vertices, weights are random real in range (1.0, 100.0)	Table 6.7	Figure 6.9 Figure 6.10
comparison of two versions of Phase I	costs equal to 1 weights equal to 1.0	Table 6.8	

Table 6.2: The experiments we conducted.

6.2.3 Experimental Results and Observations

We conduct system and functional tests for our implementations of vertex and edge separator algorithms. *System test* is to test our implementation as a whole system, *functional test* is to test each function individually. In addition, we test our implementations on a large number of planar graphs of different types against what we predict in the vertex and edge separator theorems. At last we test the efficiency of our cost-reducing technique and compare it with equi-distant levels partition. We perform our tests for the combinations of weight and cost, and list our resulting tables and figures in Table 6.2

There are several common observations that can be made for testing vertex separator of graphs with unit costs and weights (Table 6.3, Figure 6.1 and 6.2).

1. *Separator*: Costs of the obtained separators are significantly smaller than the upper bound derived in Theorem 2. This is illustrated by the ratio between the cost of the separator $c(S_t)$ and $\sqrt{\sigma(G)/t}$ (in case of unit costs, $c(S_t) = |S_t|$). By Theorem 2 this ratio is at most $4\sqrt{2} \approx 5.657$, whereas we never get a ratio bigger than 2.0 in our experiments.

2. *Balance*: Maximum weight of a component does not exceed $2tw(G)$ of test graph G with different value of t . Balance is calculated as the ratio of maximum and minimum weight of resulting components. In most cases, balance is below 2.0. The worst case happens when the graph is small and $1/t$ is very big, or when there is a small piece that can not be packed to any neighbor component. Generally speaking, our algorithm produces balanced partitions.
3. *CPU time*: Overall running times are small and do not change significantly by altering the value of t . This suggest that these algorithms are practical. Also, the heuristic of Phase III, which combines the small components into larger ones respecting the weight criteria, takes only a small fraction of time compared to the time spent in Phase I plus Phase II. Two extreme cases of the time of Phase I+II exist when $1/t = 2$. For input graph *airfoil*, *america*, *tin50000* and *tin100000*, the time of Phase I+II is longer than that when $1/t = 4, 32, 128$ because no level is found and the graph is partitioned by fundamental cycles. For input graph *trapezoid*, *whitaker*, and *crack_dual*, the time of Phase I+II is shorter because one level is found and no further partition is needed.

Similar observations can be made for experiments on graphs with unit costs and random weights in range $(1.0, 100.0)$. The results are shown in Table 6.4, Figure 6.3 and 6.4. Also, the results of experiments on graphs with random costs in range $(1, 100)$ and random weights in range $(1.0, 100.0)$ are shown in Table 6.5, Figure 6.5 and 6.6, which shows that the ratio $c(S_t)$ and $\sqrt{\sigma(G)/t}$ is much smaller since the cost of a vertex is in range of $(1, 100)$ and $\sqrt{\sigma(G)/t}$ is much bigger than $\sqrt{n/t}$. It also shows that the size of separator $|S_t|$ increases in some cases maybe because more vertices with small costs are in the separator.

Next, we make similar observations for testing edge separator of graphs with cost as vertex degree and unit weights (Table 6.6, Figure 6.7 and 6.8) as what we did for vertex separator. Denote $\sum_{v \in V} (deg(v))^2$ by $\Delta(G)$

1. *Separator*: The ratio between the size of the obtained separators $|ES_t|$ and $\sqrt{\Delta(G)/t}$ is significantly less than the constant predicted by Theorem 3. This

Graph	$\frac{1}{t}$	$\frac{ S_t }{\sqrt{n/t}}$	$ S_t $	# of pieces	min weight	max weight	time of Ph. I+II	time of Phase III	balance
trapezoid	2	0.815	33	2	387	399	0.00	0.00	1.03
$n=819$	4	0.716	41	4	189	201	0.07	0.00	1.06
$m=2286$	32	1.310	212	26	19	32	0.07	0.00	1.68
	128	1.192	386	82	2	8	0.06	0.01	4.00
tapir	2	0.376	17	2	367	640	0.10	0.00	1.74
$n=1024$	4	0.313	20	4	162	301	0.11	0.01	1.86
$m=2846$	32	1.144	207	28	14	40	0.09	0.01	2.86
	128	1.058	383	87	3	11	0.09	0.01	3.67
airfoil	2	0.434	40	2	1791	2422	0.53	0.03	1.35
$n=4253$	4	0.889	116	4	897	1314	0.58	0.03	1.46
$m=12289$	32	1.431	528	30	107	163	0.48	0.03	1.52
	128	1.511	1115	108	18	42	0.41	0.03	2.33
america	2	0.88	88	2	2122	2790	0.62	0.02	1.31
$n=5000$	4	1.245	176	4	1180	1236	0.48	0.04	1.05
$m=14787$	32	1.590	636	30	113	214	0.52	0.05	1.89
	128	1.574	1259	113	21	48	0.47	0.05	2.29
whitaker	2	0.550	77	2	4822	4901	0.06	0.08	1.02
$n=9800$	4	1.096	217	4	2139	2632	0.67	0.07	1.23
$m=28989$	32	1.627	911	31	239	336	1.14	0.08	1.41
	128	1.682	1884	114	54	104	1.09	0.09	1.93
big	2	0.951	168	2	6001	9437	2.49	0.14	1.57
$n=15606$	4	0.768	192	4	3820	3901	2.61	0.11	1.02
$m=45878$	32	1.493	1055	31	401	679	2.17	0.15	1.69
	128	1.664	2352	120	71	150	1.91	0.16	2.11
crack_dual	2	0.568	114	2	9996	10031	0.10	0.14	1.00
$n=20141$	4	0.874	248	4	4912	5036	1.13	0.13	1.03
$m=30043$	32	1.185	951	31	525	938	1.72	0.15	1.79
	128	1.231	1977	121	122	189	1.77	0.17	1.55
tin50000	2	0.705	159	2	12639	12642	4.29	0.23	1.00
$n=25440$	4	0.997	318	4	6241	6359	3.17	0.21	1.02
$m=75683$	32	1.628	1469	31	715	1103	3.88	0.23	1.54
	128	1.735	3130	122	144	208	3.44	0.24	1.44
tin100000	2	0.710	226	2	24867	25531	9.47	0.46	1.03
$n=50624$	4	0.998	449	4	12432	12656	6.68	0.46	1.02
$m=150973$	32	1.637	2083	32	1025	1580	8.67	0.46	1.54
	128	1.766	4496	124	256	437	8.47	0.48	1.71

Table 6.3: Experimental results on construction of vertex t -separator using cost-reducing technique. Testing graphs have n vertices and m edges and unit costs and weights on vertices. Time for Phase I+II and Phase III is shown separately in seconds. Number of pieces into which the graph is partitioned, minimum and maximum weight of a piece, and balance as the ratio between maximum and minimum weight of components are shown.

Figure 6.1: Graphs with unit weights and costs are partitioned by vertex t -separator. (a) trapezoid, $t = 0.5$, (b) trapezoid, $t = 0.25$, (c) tapir, $t = 0.5$, (d) tapir, $t = 0.25$

Figure 6.2: Graphs with unit weights and costs are partitioned by vertex t -separator. (a) airfoil, $t = 0.5$, (b) airfoil, $t = 0.25$, (c) america, $t = 0.5$, (d) america, $t = 0.25$

Graph	$\frac{1}{t}$	$\frac{ S_t }{\sqrt{n/t}}$	$ S_t $	# of pieces	time of Ph. I+II	time of Phase III	balance
trapezoid	2	0.815	33	2	0.00	0.01	1.03
$n=819$	4	1.048	60	4	0.07	0.01	1.86
$m=2286$	32	1.297	210	26	0.07	0.01	2.25
	128	1.127	365	74	0.06	0.00	3.50
tapir	2	0.376	17	2	0.09	0.01	1.01
$n=1024$	4	0.313	20	4	0.10	0.01	1.85
$m=2846$	32	1.011	183	29	0.07	0.01	2.36
	128	1.058	383	90	0.08	0.01	4.50
airfoil	2	0.434	40	2	0.55	0.03	1.36
$n=4253$	4	0.889	116	6	0.60	0.03	1.47
$m=12289$	32	1.456	537	30	0.47	0.04	1.42
	128	1.518	1120	109	0.37	0.04	2.30
america	2	0.88	88	2	0.70	0.04	1.31
$n=5000$	4	1.237	175	4	0.49	0.04	1.05
$m=14787$	32	1.608	643	31	0.51	0.03	1.86
	128	1.595	1276	113	0.49	0.03	2.70
whitaker	2	0.550	77	2	0.04	0.08	1.01
$n=9800$	4	1.121	222	4	0.69	0.08	1.25
$m=28989$	32	1.623	909	31	1.15	0.08	1.33
	128	1.680	1882	115	1.10	0.09	1.96
big	2	0.951	168	2	2.58	0.13	1.56
$n=15606$	4	0.768	192	4	2.57	0.13	1.02
$m=45878$	32	1.487	1051	31	2.24	0.15	1.67
	128	1.669	2359	121	1.96	0.15	1.94
crack_dual	2	0.568	114	2	0.11	0.15	1.01
$n=20141$	4	0.874	248	4	1.08	0.14	1.04
$m=30043$	32	1.191	956	32	1.76	0.15	1.70
	128	1.231	1977	121	1.76	0.16	1.67
tin50000	2	0.705	159	2	4.33	0.20	1.00
$n=25440$	4	0.997	318	4	3.27	0.22	1.02
$m=75683$	32	1.645	1484	31	3.79	0.24	1.40
	128	1.730	3129	121	3.45	0.25	1.80
tin100000	2	0.704	224	2	9.05	0.43	1.00
$n=50624$	4	0.998	449	4	6.62	0.45	1.01
$m=150973$	32	1.671	2127	31	8.57	0.47	1.58
	128	1.765	4493	123	8.32	0.49	1.33

Table 6.4: Experimental results on constructing vertex t -separator of graphs with unit costs and random weights in range (1.0, 100.0) on vertices. Cost-reducing technique is used. Time for Phase I + II and Phase III is shown separately in seconds. Number of pieces into which the graph is partitioned, and balance are shown.

Figure 6.3: Graphs with random weights in range (1.0, 100.0) and unit costs are partitioned by vertex t -separator. (a) trapezoid, $t = 0.5$, (b) trapezoid, $t = 0.25$, (c) tapir, $t = 0.5$, (d) tapir, $t = 0.25$

Figure 6.4: Graphs with random weights in range (1.0,100.0) and unit costs are partitioned by vertex t -separator. (a) airfoil, $t = 0.5$, (b) airfoil, $t = 0.25$, (c) america, $t = 0.5$, (d) america, $t = 0.25$

Graph	$\frac{1}{t}$	$\frac{c(S_t)}{\sqrt{\sigma(G)/t}}$	$ S_t $	# of pieces	time of Ph. I+II	time of Phase III	balance
trapezoid	2	0.192	15	2	0.09	0.00	1.04
$n=819$	4	0.491	63	4	0.07	0.00	1.69
$m=2286$	32	1.07	224	25	0.12	0.01	2.13
	128	0.972	367	75	0.13	0.02	3.02
tapir	2	0.282	18	2	0.14	0.00	1.75
$n=1024$	4	0.269	35	4	0.12	0.01	1.08
$m=2846$	32	0.839	197	27	0.12	0.00	2.77
	128	0.904	363	100	0.12	0.01	9.33
airfoil	2	0.487	97	2	0.60	0.04	1.01
$n=4253$	4	0.427	115	4	0.65	0.04	1.24
$m=12289$	32	1.088	591	32	0.69	0.03	1.95
	128	1.325	1246	105	0.71	0.04	2.17
america	2	0.495	113	2	0.90	0.04	1.29
$n=5000$	4	0.400	120	4	0.75	0.03	1.68
$m=14787$	32	1.101	726	30	0.85	0.05	2.15
	128	1.211	1317	103	0.85	0.06	2.14
whitaker	2	0.242	78	2	1.72	0.09	1.01
$n=9800$	4	0.629	290	4	1.94	0.09	1.37
$m=28989$	32	1.398	1166	33	1.78	0.09	1.85
	128	1.445	2286	114	1.83	0.10	2.31
big	2	0.504	201	2	4.07	0.14	1.49
$n=15606$	4	0.504	288	4	4.16	0.14	1.12
$m=45878$	32	0.906	1148	34	4.10	0.14	1.75
	128	1.184	2775	125	4.08	0.15	2.87
crack_dual	2	0.200	95	2	2.85	0.14	1.07
$n=20141$	4	0.561	237	4	2.67	0.15	1.19
$m=30043$	32	0.976	997	33	3.46	0.16	1.96
	128	1.016	2037	130	3.83	0.18	2.55
tin50000	2	0.310	170	2	4.76	0.23	1.69
$n=25440$	4	0.752	602	4	5.02	0.23	1.37
$m=75683$	32	1.292	2074	33	6.02	0.23	1.96
	128	1.469	4219	130	6.05	0.25	2.22
tin100000	2	0.302	246	2	9.77	0.48	1.11
$n=50624$	4	0.563	638	4	9.91	0.49	1.31
$m=150973$	32	1.324	3258	35	13.94	0.47	1.79
	128	1.479	6345	130	14.05	0.49	2.38

Table 6.5: Experimental results on constructing vertex t -separator of graphs with random integer costs in range (1, 100), and random weights in range (1.0, 100.0) on vertices. cost-reducing technique is used. $c(S_t)$ is the cost of separator S_t . Time for Phase I + II and Phase III is shown separately in seconds. Number of pieces into which the graph is partitioned, and balance are shown.

Figure 6.5: Graphs with random weights in range (1.0, 100.0) and random integer costs in range (1, 100) are partitioned by vertex t -separator. (a) trapezoid, $t = 0.5$, (b) trapezoid, $t = 0.25$, (c) tapir, $t = 0.5$, (d) tapir, $t = 0.25$

Figure 6.6: Graphs with random weights in range (1.0, 100.0) and random integer costs in range (1, 100) are partitioned by vertex t -separator. (a) airfoil, $t = 0.5$, (b) airfoil, $t = 0.25$, (c) america, $t = 0.5$, (d) america, $t = 0.25$

ratio is at most $4\sqrt{2} \approx 5.657$ in Theorem 3, whereas we never get this ratio bigger than 1.0 in our experiments.

2. *Balance*: Maximum weight of a component does not exceed $2tw(G)$ of test graph G with different value of t . Balance is calculated as the ratio of maximum and minimum weight of resulting components. In most cases, balance is below 2.0. The worst case happens when the graph is small and $1/t$ is very big, or when a vertex in vertex separator can not be packed to its neighbor components and it forms a new component by itself. Generally speaking, our algorithm produces balanced partitions.
3. *CPU time*: Time for computing edge separators is small, and does not vary substantially by varying the value of t . Time for Phase III is significantly smaller than the time for Phase I+II for most values of t . For different values of t , time for both Phase I+II and Phase III is stable for the same graph.

The results of edge-separator experiments on graphs with cost as vertex degree and random weight in range (1.0, 100.0) are shown in Table 6.7, Figure 6.9 and 6.10. These results are very similar to the results when testing graphs with unit weight.

At last, we conduct experiments on the efficiency of the cost-reducing technique from Chapter 4. We have two implementations of Phase I in our algorithm. The first version of Phase I applies equi-distant levels partition. The second version employs the cost-reducing technique. We test the two versions of Phase I for unit costs only. For our convenience, we compare the CPU time of Phase I+II and the cost of the separator generated by Phase I+II in the experiments and present results in Table 6.8. Figure 6.11 shows the improvement in cost of separators by ratio between cost of separators generated by cost-reducing technique and cost of separators generated by equi-distance levels partition. If the ratio is more close to 1.0 or greater than 1.0, there is little or no improvement. If the ratio is far less than 1.0, there is more improvement. We observe the following:

1. *CPU time*: The computation of t -separators using cost-reducing technique takes

slightly more time compared to the equi-distant levels partition. These results are consistent with varying the parameter t . There is one special case for $1/t = 4$, optimized-time is smaller than equi-time of Phase I+II for graphs *whitaker*, *america*, and *tin50000*. This is because the equi-distance level approach finds more levels than the cost-reducing technique, and it takes a little more time to compute them. For instance, for *whitaker*, the equi-distance level approach finds 6 levels and the cost-reducing technique finds 2 levels; for *america*, the equi-distance computes 3 levels and the cost-reducing technique selects 2; for *tin50000*, the equi-distance computes 3 levels and the cost-reducing technique computes 1 level only.

2. *Improvement in cost of separator*: Generally speaking, when $1/t = 64, 128, 256$, the improvement in cost of separators using cost-reducing technique are stable and vary from 20% to 35% except in graph *big* it is about 15%. Graph *trapezoid* and *tapir* have gradually less improvements simply because they're relatively small graphs. Most of the test graphs have their significant improvement in the range [35%, 55%] when $1/t = 4$ except *airfoil* and *big* have very small improvement about 5%, respectively. Graph *america* has negative improvement when $1/t = 4$ because the cost of separator generated by equi-distant level approach is less than that generated by the cost reducing technique. The reason is that the equi-distant level approach finds 3 levels to partition the graph into 4 components but the cost-reducing technique finds 2 levels and a fundamental-cycle is computed to partition the graph further, which results in more vertices in the vertex separator compared to the separator generated by the equi-distance level approach.

6.2.4 Comparison

We briefly compare our edge separator results with Gilbert et al [16], introduced in Section 2.2.3.1 in Chapter 2. Table 6.9 presents this comparison for graph *tapir* because we do not have the other graphs listed in [16]. In [16], Matlab is used to

Graph	$\frac{1}{t}$	$\frac{ ES_t }{\sqrt{\Delta(G)/t}}$	$ ES_t $	# of pieces	min weight	max weight	time of Ph. I+II	time of Phase III	balance
trapezoid	2	0.122	28	2	404	415	0.10	0.00	1.03
$n=819$	4	0.338	110	4	137	257	0.10	0.00	1.88
$m=2286$	32	0.485	447	27	17	51	0.09	0.01	3.00
	128	0.498	918	98	1	12	0.09	0.01	12.00
tapir	2	0.073	20	2	487	537	0.13	0.00	1.10
$n=1024$	4	0.231	89	4	244	263	0.12	0.01	1.08
$m=2846$	32	0.433	472	29	25	48	0.10	0.01	1.92
	128	0.489	1066	118	1	16	0.10	0.01	16.00
airfoil	2	0.149	80	2	1778	2475	0.67	0.03	1.39
$n=4253$	4	0.295	224	4	899	1388	0.60	0.04	1.54
$m=12289$	32	0.495	1064	31	111	196	0.61	0.03	1.77
	128	0.547	2350	112	17	63	0.55	0.03	3.71
america	2	0.342	205	2	2089	2911	0.74	0.04	1.39
$n=5000$	4	0.374	317	4	1020	1392	0.55	0.04	1.36
$m=14787$	32	0.587	1410	31	96	286	0.67	0.05	2.98
	128	0.573	2751	113	27	64	0.62	0.05	2.37
whitaker	2	0.231	192	2	4854	4946	0.23	0.09	1.02
$n=9800$	4	0.364	428	4	2398	2534	1.52	0.08	1.06
$m=28989$	32	0.566	1883	31	221	409	1.43	0.08	1.85
	128	0.600	3992	122	44	129	1.31	0.09	2.93
big	2	0.302	315	2	6134	9472	3.54	0.15	1.54
$n=15606$	4	0.270	398	4	3881	3934	2.73	0.14	1.01
$m=45878$	32	0.514	2144	34	308	514	3.19	0.14	1.67
	128	0.581	4851	125	65	201	2.92	0.16	3.09
crack_dual	2	0.179	107	2	10046	10095	0.46	0.15	1.00
$n=20141$	4	0.281	238	4	4798	5224	1.60	0.15	1.09
$m=30043$	32	0.414	992	32	484	915	2.40	0.15	1.89
	128	0.455	2180	125	89	223	2.36	0.18	2.51
tin50000	2	0.235	316	2	12561	12879	4.81	0.23	1.03
$n=25440$	4	0.378	718	4	6312	6389	4.30	0.23	1.01
$m=75683$	32	0.597	3207	31	660	1062	4.94	0.23	1.61
	128	0.617	6639	124	110	307	4.64	0.25	2.79
tin100000	2	0.235	447	2	24504	26120	9.65	0.48	1.07
$n=50624$	4	0.379	1017	4	12559	12763	9.42	0.49	1.02
$m=150973$	32	0.598	4545	32	1245	1794	11.60	0.49	1.44
	128	0.624	9482	125	264	669	11.45	0.51	2.53

Table 6.6: Experimental results on construction of edge t -separators with cost-reducing technique. Edge t -separator ES_t is computed from vertex t -separator where unit weight and cost as degree of a vertex are associated with vertices, respectively. Times for Phase I + II and Phase III are shown separately in seconds. Number of pieces into which the graph is partitioned, minimum and maximum weight of a piece, and balance are shown.

Figure 6.7: Graphs with unit weights and costs as degree of vertices are partitioned by edge t -separator. (a) trapezoid, $t = 0.5$, (b) trapezoid, $t = 0.25$, (c) tapir, $t = 0.5$, (d) tapir, $t = 0.25$

Figure 6.8: Graphs with unit weights and costs as degree of vertices are partitioned by edge t -separator. (a) airfoil, $t = 0.5$, (b) airfoil, $t = 0.25$, (c) america, $t = 0.5$, (d) america, $t = 0.25$

Graph	$\frac{1}{t}$	$\frac{ ES_t }{\sqrt{\Delta(G)/t}}$	$ ES_t $	# of pieces	time of Ph. I+II	time of Phase III	balance
trapezoid	2	0.122	28	2	0.10	0.00	1.03
$n=819$	4	0.282	92	4	0.09	0.00	1.04
$m=2286$	32	0.507	467	29	0.08	0.01	2.26
	128	0.493	909	96	0.12	0.01	18.80
tapir	2	0.073	20	2	0.13	0.01	1.09
$n=1024$	4	0.296	114	4	0.11	0.01	1.39
$m=2846$	32	0.434	473	30	0.10	0.01	2.43
	128	0.488	1065	113	0.10	0.01	13.12
airfoil	2	0.149	80	2	0.67	0.04	1.40
$n=4253$	4	0.306	232	4	0.61	0.03	1.55
$m=12289$	32	0.489	1050	30	0.57	0.04	1.67
	128	0.548	2356	110	0.58	0.03	4.69
america	2	0.342	205	2	0.78	0.04	1.39
$n=5000$	4	0.374	317	4	0.55	0.04	1.39
$m=14787$	32	0.560	1343	30	0.66	0.04	2.39
	128	0.574	2755	116	0.64	0.05	3.80
whitaker	2	0.226	188	2	0.25	0.08	1.02
$n=9800$	4	0.365	429	4	1.59	0.08	1.04
$m=28989$	32	0.575	1911	32	1.44	0.09	1.91
	128	0.601	4000	123	1.33	0.10	3.53
big	2	0.302	315	2	3.54	0.14	1.54
$n=15606$	4	0.270	398	4	2.77	0.14	1.01
$m=45878$	32	0.505	2110	32	3.18	0.14	1.86
	128	0.576	4810	125	2.96	0.16	2.56
crack_dual	2	0.175	105	2	0.46	0.16	1.00
$n=20141$	4	0.281	238	4	1.56	0.16	1.09
$m=30043$	32	0.412	987	32	2.37	0.16	1.91
	128	0.451	2162	124	2.38	0.17	2.60
tin50000	2	0.236	317	2	5.10	0.23	1.02
$n=25440$	4	0.449	854	4	4.33	0.22	1.03
$m=75683$	32	0.592	3185	32	5.01	0.23	1.68
	128	0.618	6649	124	4.57	0.25	2.32
tin100000	2	0.235	447	2	9.65	0.49	1.04
$n=50624$	4	0.378	1016	4	9.44	0.49	1.02
$m=150973$	32	0.597	4540	32	11.63	0.49	1.58
	128	0.621	9438	125	11.47	0.52	2.50

Table 6.7: Experimental results on construction of edge t -separators with cost-reducing technique. Edge t -separator ES_t is computed from vertex t -separator where weight as random real in range (1.0, 100.0) and cost as degree of a vertex are associated with vertices, respectively. Times for Phase I + II and Phase III are shown separately in seconds. Number of pieces into which the graph is partitioned, and balance are shown.

Figure 6.9: Graphs with random weights in range (1.0, 100.0) and costs as degree of vertices are partitioned by edge t -separator. (a) trapezoid, $t = 0.5$, (b) trapezoid, $t = 0.25$, (c) tapir, $t = 0.5$, (d) tapir, $t = 0.25$

Figure 6.10: Graphs with random weights in range (1.0, 100.0) and costs as degree of vertices are partitioned by edge t -separator. (a) airfoil, $t = 0.5$, (b) airfoil, $t = 0.25$, (c) america, $t = 0.5$, (d) america, $t = 0.25$

Graph	1/t	4	16	32	64	128	256
trapezoid	equi-cost	80	217	336	453	560	599
$n=819$	optimized-cost	43	156	235	322	447	533
$m=2286$	equi-time	0.04	0.06	0.05	0.04	0.02	0.01
	optimized-time	0.07	0.07	0.07	0.06	0.06	0.05
tapir	equi-cost	82	202	328	470	595	665
$n=1024$	optimized-cost	49	158	260	354	489	583
$m=2846$	equi-time	0.07	0.07	0.07	0.05	0.04	0.01
	optimized-time	0.11	0.09	0.09	0.09	0.09	0.08
airfoil	equi-cost	125	486	744	1084	1563	2196
$n=4253$	optimized-cost	120	398	585	842	1191	1624
$m=12289$	equi-time	0.55	0.44	0.39	0.36	0.32	0.25
	optimized-time	0.58	0.48	0.48	0.46	0.41	0.40
america	equi-cost	153	536	834	1222	1721	2409
$n=5000$	optimized-cost	189	459	720	993	1376	1875
$m=14787$	equi-time	0.60	0.50	0.45	0.45	0.38	0.34
	optimized-time	0.48	0.53	0.52	0.48	0.47	0.46
whitaker	equi-cost	360	829	1254	1791	2504	3532
$n=9800$	optimized-cost	238	632	925	1390	1932	2669
$m=28989$	equi-time	0.87	1.00	0.98	0.99	0.94	0.88
	optimized-time	0.67	1.14	1.14	1.12	1.09	1.09
big	equi-cost	208	845	1406	1998	2938	4190
$n=15606$	optimized-cost	198	682	1108	1709	2471	3501
$m=45878$	equi-time	2.51	2.09	2.00	1.86	1.66	1.46
	optimized-time	2.61	2.58	2.17	2.09	1.91	1.91
crack_dual	equi-cost	418	1070	1651	2346	3422	4830
$n=20141$	optimized-cost	259	737	1088	1611	2301	3272
$m=30043$	equi-time	0.87	1.31	1.64	1.74	1.71	1.36
	optimized-time	1.13	1.72	1.72	1.73	1.77	1.75
tin50000	equi-cost	711	1363	1991	2853	4074	5711
$n=25440$	optimized-cost	319	962	1508	2216	3175	4446
$m=75683$	equi-time	4.04	3.51	3.28	3.18	2.86	2.60
	optimized-time	3.17	3.75	3.88	3.58	3.44	3.25
tin100000	equi-cost	788	1910	2806	4091	5804	8245
$n=50624$	optimized-cost	450	1344	2118	3119	4582	6447
$m=150973$	equi-time	6.46	7.81	7.95	7.54	6.98	6.37
	optimized-time	6.68	8.66	8.67	8.59	8.47	8.12

Table 6.8: Equi-distant levels vs. optimized choice of levels. We denote by *equi-cost* and *optimized-cost* the *cost of separators* using equi-distant levels, and optimized choice of levels chosen by cost-reducing technique, respectively. We denote *equi-time* and *optimized-time* the computation time of Phase I+II, respectively. All times are in seconds.

(a)

(b)

Figure 6.11: Both (a) and (b) demonstrate the improvement in costs of vertex t -separator constructed by cost-reducing technique vs. equi-distant levels partition, based on the experimental results shown in Table 6.8. x -axis represents $1/t$ and $1/t = 4, 16, 32, 64, 128, 256$. y -axis is the ratio between optimized-cost and equi-cost. If the ratio is close to 1.0 or greater than 1.0, there is no improvement. If the ratio is far less than 1.0, there is improvement.

partitions	FEM(2D)	Spectral	Coordinate Bisection	Default Geometric	Best Geometric	Our algorithm
2-way ($1/t = 2$)	tapir	59	55	37	32	20
128-way ($1/t = 128$)	tapir	1278	1387	1239		1066

Table 6.9: Compare our edge separator results with Gilbert et al[16]. In our case, edge separator is generated from graph *tapir* with unit weights and costs as degree of vertices.

implement coordinate bisection. Hendrickson and Leland’s Chaco package [31] are also used to find the spectral bisection. A parameter to the geometric algorithm is the number of random trials of great circles to be made. The “default geometric” column reports results for 30 trials and “best geometric” reports the results for 7000 trials. Our experimental results show that our planar graph partitioning algorithms are competitive with other modern graph partitioning methods.

METIS, introduced in Section 2.2.3.4 in Chapter 2, is a software implementation due to Karypis and Kumar [25, 23, 26]. This software package is mainly for partitioning large irregular graphs and large meshes, and computing fill-reducing orderings of sparse matrices. The algorithms in METIS are based on multilevel graph partitioning described in [25, 23, 26]. The various programs in METIS require as input either a file storing a graph or a file storing a mesh. METIS defined a couple of formats for input graph or mesh based on whether weight is associated with vertices, or edges, or both, or no weight with vertices and edges. We do not compare our results with METIS mainly because our approach is totally different from METIS and the results may not be comparable.

In conclusion, this chapter presents the design issues in terms of implementation, followed by experiment details, experimental results and our observation. Finally, it presents the comparison of our results with Geometric Mesh Partitioning by Gilbert et al. The experimental results show that our planar graph partitioning algorithms are efficient and effective. They are competitive with other modern graph partitioning methods.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this chapter, we summarize new graph separator results and methods that can be used to improve the speed of graph partitioning and/or to improve the size of the constructed separator. The main results of this thesis have appeared in the workshop on *Algorithm Engineering and Experiments* (Alenex) 2002[1].

The main results of this thesis are:

1. We show that there exists a vertex t -separator of G whose cost is at most $4\sqrt{2\sigma(G)/t}$, where $\sigma(G) = \sum_{v \in V} (c(v))^2$, and provide an algorithm for constructing such a separator. The cost of the separator is asymptotically optimal for the class of planar graphs. The running time of the algorithm is $O(n + T_{\text{SSSP}}(G))$, where $T_{\text{SSSP}}(G)$ is the time for computing single source shortest path tree from a fixed vertex in G .
2. We present a technique for tuning the above algorithm that results in considerable reduction of the cost of the produced separators. The computational cost of this technique is analyzed theoretically in the case of integer costs and experimentally tested.

3. We show that there exists an edge t -separator of at most $4\sqrt{2\Delta(G)/t}$, where $\Delta(G) = \sum_{v \in V} (\deg(v))^2$ and $\deg(v)$ is the degree of a vertex v . The size of the edge separator is asymptotically optimal for the class of planar graphs. Our algorithm constructs such a separator in $O(n + T_{\text{SSSP}}(G))$ time.
4. We implement our algorithms and support the theoretical results by extensive experiments. Unlike other implementations of graph partitioning algorithms based on heuristics, ours always produce balanced partitions and separators whose worst-case cost (size) is guaranteed. Moreover, the experiments suggest that the constants in the estimates on the cost of separators can be improved. In addition, experiments indicate that our tuning technique is effective and computationally efficient.

Result 1 is a generalization of many existing results on separators. By appropriately choosing weight, cost and the parameter t we show how to obtain a variety of known results in Chapter 5. In most of the existing work, the parameter t is chosen to be $2/3$ and a fast (usually linear time) algorithm is presented to obtain such a partition. Partition with smaller values of t is achieved by recursively applying the algorithm, and hence has an extra factor of $O(\log(1/t))$ in the complexity. Result 1 works for to any value of t and the time for finding a t -separator does not depend on t . This is supported by our experiments. Moreover, our experiments show that the execution time is consistently small for different values of t and hence our results are not only theoretically but also practically attractive.

At last, we briefly compare our edge separator results with Gilbert et al [16] in Chapter 6. The experiments show that our algorithms are competitive with other modern graph partitioning methods.

7.2 Limitations and Future Work

One of the limitations in our algorithms is that our programs sometimes can not produce the exact number of pieces from input graphs, as user asks. This limitation

is from the nature of the algorithms. In Phase I, we first try to select a set of levels to partition input graph, then in Phase II, we use fundamental cycles to partition “heavy” components further. These two phases lead to the situation that an input graph may be partitioned into many small piece under the weight condition $tw(G)$ (or $\alpha tw(G)$ where α is a constant chosen in range $[1.0, 2.0]$). In Phase III when we try to pack a smaller piece to its neighboring component, there are two possibilities: it can either be absorbed to its neighboring component successfully, or forms its own region. In this phase, we did not consider the possibility of merging more than two neighboring components into one at the same time. So, Phase III needs to be improved in the future.

Another limitation is in the way we construct edge separator from vertex separator. The size of edge separator is not optimized even though we used “best fit” then “first fit” strategy and relaxed the weight condition to $2tw(G)$. There might be other way to optimize the size of edge separator.

For the limited time of this research, we do not compare our results with METIS[25, 23, 26] which has implementation and experimental results published. There are two reasons: one is we use a different graph format from what METIS uses. The other reason is that we have different approach from METIS and the results may not be comparable. Our approach is to construct a vertex separator first, then to obtain edge separator. METIS is based on graph growing heuristics and spectral bisection, and it computes edge separator directly.

Another possible work is to implement Kernighan and Lin algorithm [36] in the Phase III of our algorithm. It may optimize our separators further.

Bibliography

- [1] L. Aleksandrov, H. Djidjev, H. Guo, and A. Maheshwari, Partitioning planar graphs with costs and weights, *ALLENEX*, San Francisco, Jan. 2002
- [2] L. Aleksandrov and H. Djidjev, Linear Algorithms for partitioning embedded graphs of bounded genus, *SIAM J. Disc. Math.*, 9(1):129-150, 1996
- [3] S.N. Bhatt and F.T. Leighton, A framework for solving VLSI graph layout problems, *Journal of computer and System Sciences*, 28:300-343, 1984
- [4] W.J. Camp, S.J. Plimpton, B.A. Hendrickson, and R.W. Leland, Massively parallel methods for engineering and science problems, *Communications of the ACM*, 37(4):30-41, Apr. 1994
- [5] Tony F. Chan, P. Ciarlet Jr., and W.K. Szeto, On the Optimality of the Median Cut Spectral Bisection Graph Partitioning Method, *SIAM Journal on Scientific Computing*, 18(3):943-948, 1997
- [6] E.G. Coffman Jr., M.R. Garey, and D.S. Johnson, Approximation Algorithms for Bin-Packing – An Updated Survey, *Algorithm Design for Computer System Design*, ed. by Ausiello, Lucertini, and Serafini. Springer-Verlag, 1984
- [7] K. Diks, H.N. Djidjev, O. Sykora, and I. Vrto, Edge separators of planar graphs and outerplanar graphs with applications, *J. Algorithms*, 34:258-279, 1993
- [8] H.N. Djidjev, Grammati E. Pantziou, and Christos D. Zaroliagis, Improved algorithms for dynamic shortest paths, *Algorithmica*, 28(4):367-389, 2000

- [9] H.N. Djidjev, Partitioning planar graphs with vertex costs: Algorithms and applications, *Algorithmica*, 28(1):51-57, 2000
- [10] H.N. Djidjev, Linear algorithms for graph separation problems, *Proceedings of 1st Scandinavian Workshop on Algorithm Theory*, pp.216-222, 1988
- [11] H.N. Djidjev, On the problem of partitioning planar graphs, *SIAM Journal on Algebraic and Discrete Methods*, 3:229-240, 1982
- [12] H.N. Djidjev, A separator theorem, *Compt. rend. Acad. bulg. Sci.*, 34:643-645, 1981
- [13] L. Danzer, J. Fonlupt, and V. Klee, Helly's theorem and its relatives, *Proceedings of Symposia in Pure Mathematics, American Mathematical Society*, 7:101-180, 1963
- [14] G.N. Frederickson, Fast algorithm for shortest paths in planar graphs, with applications, *SIAM J. Computing*, 16:1004-1022, 1987
- [15] Michael T. Goodrich, Planar separators and parallel polygon triangulation, *Proceedings of 24th Symp. on Theory of Computing*, pp.507-516, 1992
- [16] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng, Geometric mesh partitioning: Implementation and experiments *SIAM Journal on Scientific Computing*, 19(6):2091-2110, 1998
- [17] J.R. Gilbert, E.G. Ng, and B.W. Peyton, Separators and structure prediction in sparse orthogonal factorization, *Technical Report CSL-93-15, Xerox Corporation, Palo Alto Research Center*, 1993
- [18] J.R. Gilbert and R. Schreiber, Highly parallel sparse cholesky factorization, *SIAM J. Scientific and Statistical Computing*, 13:1151-1172, 1992
- [19] John R. Gilbert, Joan P. Hutchinson, and Robert E. Tarjan, A separator theorem for graphs of bounded genus, *J. Algorithms*, 5:391-407, 1984

- [20] John R. Gilbert, Donald J. Rose, and Anders Edenbrandt, A separator theorem for chordal graphs, *SIAM Journal on Algebraic and Discrete Methods*, pp.306-313, 1984
- [21] John R. Gilbert, Graph separator theorems and sparse Gaussian elimination, *Ph.D. thesis, Stanford Universit*, Stanford, CA, 1980
- [22] Camille Jordan, Sur les assemblages de lignes, *Journal Reine Angew. Math.*, 70:185-190, 1869
- [23] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *Technical Report TR 95-037, Dept. of Computer Science, Univ. of Minnesota*, 1995
- [24] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *Technical Report TR 95-035, Dept. of Computer Science, Univ. of Minnesota*, 1995
- [25] G. Karypis and V. Kumar, Metis 3.0: Unstructured graph partitioning and sparse matrix ordering system, *Technical Report 97-061, Dept. of Computer Science, Univ. of Minnesota*, 1997
- [26] G. Karypis and V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, *Technical Report TR 95-064, Dept. of Computer Science, Univ. of Minnesota*, 1995
- [27] C. Kuratowski, Sur le problème des corbes gauches en topologie, *Fund. Math.*, 15:271-283, 1930
- [28] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969
- [29] D.W. Hall and G. Spencer, *Elementary Topology*, John Wiley, New York, 1955
- [30] Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian, Faster shortest-path algorithms for planar graphs, *Journal of Computer and System Sciences*, 55(1):3-23, Aug. 1997

- [31] B. Hendrickson and R. Leland, The Chaco user's guide - version 2.0, *Technical Report SAND94-2692, Sandia National Laboratories*, 1994
- [32] B. Hendrickson and R. Leland, An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations, *Technical Report SAND92-1460, Sandia National Laboratory, Albuquerque, NM*, Sept. 1992
- [33] Charles E. Leiserson, Area-efficient graph layouts (for VLSI), *Proc. 21st Annual IEEE Symposium on Foundations of Computer Science*, pp.270-281, 1980
- [34] P.M. Lewis II, R. E. Stearns and J. Hartmanis, Memory bounds for the recognition of context-free and context-sensitive languages, *IEEE Conference Record on Switching Theory and Logical Design*, pp.191-202, 1965
- [35] David S. Johnson, Fast Algorithms for Bin Packing, *Journal of Computer and System Sciences* 8:272-314, 1974
- [36] B.W. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs *The Bell System Technical Journal*, pp.291-307, Feb. 1970
- [37] C.E. Leiserson, Area efficient VLSI computation, In *Foundations of Computing*, MIT Press, Cambridge, MA, 1983
- [38] R.J. Lipton and R.E. Tarjan, A separator theorem for planar graphs, *SIAM J. Appl. Math.*, 36(2):177-189, 1979
- [39] R.J. Lipton and R.E. Tarjan, Applications of a planar separator theorem, *SIAM J. Computing*, 9(3):615-627, 1980
- [40] H.S. Maini, K.G. Mehrotra, C.K. Mohan, S. Ranka, Genetic algorithms for graph partitioning and incremental graph partitioning, *CRPC-TR-94504, Rice University*, 1994
- [41] Kurt Mehlhorn and Stefan Näher, LEDA, a Platform for Combinatorial and Geometric Computing, *Communications of ACM*, 38:96-102, 1995

- [42] Gary L. Miller, Shang-Hua Teng, Willian Thurston, and Stephen A. Vavasis, Finite element meshes and geometric separators, *SIAM J. Scientific Computing*, to appear 1996
- [43] Gary L. Miller, Shang-Hua Teng, Willian Thurston, and Stephen A. Vavasis, Automatic mesh partitioning, In A. George, J. Gilbert, and J. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, IMA Volumes in Mathematics and its Applications. Springer-Verlag, 1993
- [44] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis, A unified geometric approach to graph separators, *Proceedings of the 32nd FOCS*, pp.538-547, 1991
- [45] Gary L. Miller and Willian Thurston, Separators in two and three dimensions, *Proceedings of the 22nd Symp. on Theory of Computing*, pp.300-309, 1990
- [46] A. Pothen, H.D. Simon, and K.-P. Liou, Partitioning sparse matrices with eigenvectors of graphs, *SIAM J. Matrix Anal. Appl.*, 11(3):430-452, Jul. 1990
- [47] M.S. Paterson and C.E. Hewitt, Comparative schematology, Record of Project MAC Conf. on *Concurrent Systems and Parallel Computation*, pp.119-128, 1970
- [48] W.J. Paul, R.E. Tarjan, and J.R. Celoni, Space bounds for a game on graphs, *Math. Systems Theory*, 10:239-251, 1977
- [49] K. Scholegel, G. Karypis, and V. Kumar, Graph Partitioning for High Performance Scientific Simulations, In J. Dongarra et al., editor, *CRPC Parallel Computing Handbook*. Morgan Kaufmann (in press).
- [50] H. Simon, Partitioning of unstructured problems for parallel processing, *Comput. Systems Engrg.*, 2:135-148, 1991
- [51] R. Sethi, Complete register allocation problems, *SIAM J. Comput.*, 4:226-248, 1975
- [52] R. Williams, Performance of dynamic load balancing algorithms for unstructured mesh calculations, *Concurrency*, 3:457-481, 1991