EXPLORING ALGORITHMS FOR BRANCH DECOMPOSITIONS OF PLANAR GRAPHS

A thesis presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Hiep Dinh
November 2008
© 2008

Hiep Dinh

All Rights Reserved
This thesis entitled
EXPLORING ALGORITHMS FOR BRANCH DECOMPOSITIONS OF PLANAR GRAPHS

by
HIEP DINH

has been approved for
the Department of Computer Science
and the Russ College of Engineering and Technology by

David W. Juedes
Associate Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
Abstract

DINH, HIEP, M.S., November 2008, Computer Science

Exploring algorithms for branch decompositions of planar graphs (132 pp.)

Director of Thesis: David W. Juedes

A branch decomposition is a type of graph decomposition closely related to the widely studied tree decompositions introduced by Robertson and Seymour. Unlike tree decompositions, optimal branch decompositions and the branch-width of planar graphs can be computed in polynomial time. The ability to construct optimal branch decompositions in polynomial time leads to efficient solutions for generally hard problems on instances restricted to planar graphs.

This thesis studies efficient algorithms for computing optimal branch decompositions for planar graphs. Our main contribution is an improved software package for graph decompositions with efficient implementations of two additional decomposition classes: carving decompositions and branch decompositions. Polynomial time solutions for INDEPENDENT-SET on general graphs using path decompositions, tree decompositions, and branch decompositions with bounded width are also explored as examples of how graph decompositions can be used to solve NP-HARD problems.

Approved:

David W. Juedes

Associate Professor of Electrical Engineering and Computer Science
Acknowledgments

I am eternally grateful to my thesis advisor, Dr. David Juedes, for his guidance, advice, encouragement, and constant support during my study at Ohio University. I wish to thank him for providing the graph decompositions software as a foundation for the implementations in my thesis. I am thankful to my thesis committee members Dr. Cynthia Marling, Dr. David Chelberg, and Dr. Xiaoping Shen. I would like to thank everyone who helped me review my thesis for their valuable suggestions. I am greatly in debt to my family for their endless supports without which this work could not have been completed.
# Table of Contents

Abstract .......................................................... 4  
Acknowledgments .......................................................... 5  
List of Figures .......................................................... 8  
List of Tables .......................................................... 9  
List of Algorithms ..................................................... 10  

1 Introduction ....................................................... 11  
1.1 Background ......................................................... 12  
1.2 Related Work ...................................................... 15  
1.3 Contributions of This Thesis ........................................ 16  
1.4 Thesis Overview .................................................... 17  

2 NP-Complete Problems and Polynomial-time Algorithms ............... 19  
2.1 A Traditional Look at NP-Complete Problems .......................... 19  
2.2 Fixed Parameter Tractability and Polynomial-time Algorithms for NP-Complete Problems ........................................... 20  
2.3 Fixed Parameter Intractability and the W Hierarchy .................. 22  

3 Graph Decompositions and FPT Algorithms .......................... 30  
3.1 Path Decompositions ................................................ 31  
3.2 Tree Decompositions ............................................... 36  
3.3 Branch Decompositions ............................................. 42  
3.4 Tree Decompositions and Branch Decompositions ..................... 54  

4 Optimal Branch Decompositions for Planar Graphs ....................... 60  
4.1 Preliminaries ....................................................... 61  
4.2 The Carving-width Decision Problem and the Rat-catching Game ..... 69  
4.3 The Optimization Problem .......................................... 80  
4.4 The Constructive Algorithm ......................................... 82  

5 Implementations ..................................................... 86  
5.1 Software Overview ................................................ 86  
5.2 Implementations .................................................... 88  
5.3 Experimental Results ............................................... 99
6  Conclusions and Future Work ...................................... 103

Bibliography ..................................................................... 106

A  Source Code ................................................................. 111
   A.1 Class Carving Decomposition .................................... 111
   A.2 Class Branch Decomposition .................................... 124
   A.3 Helper Functions ....................................................... 129
List of Figures

2.1 Converting a graph to a circuit for Independent-Set ........................................ 25
2.2 Converting a graph to a circuit for Dominating-Set ........................................ 26

3.1 A branch decomposition of width three .............................................................. 44
3.2 Midsets of three edges incident to the same vertex on a branch decom-
position .................................................................................................................. 45
3.3 Vertex distributions on midsets ............................................................................ 47
3.4 Converting a branch decomposition to a rooted branch decomposition ............... 49
3.5 3 nodes separate a branch decomposition tree into 7 disjoint trees .................... 55

4.1 Different embeddings for the same graph ........................................................... 61
4.2 The same graph has two dual graphs that are not isomorphic ......................... 62
4.3 An example of how to construct medial graph .................................................. 64
4.4 Two edges incident to a vertex of degree 2 in the original graph ....................... 66
4.5 Two edges incident to a vertex of degree > 2 in the original graph .................... 67
4.6 Multiple edges in the original graph .................................................................. 68
4.7 The rat-catching game happens on the black graph ........................................... 71
4.8 The rat-catching game with noisy edges marked as red ...................................... 72
4.9 An example of the rat-catching game .................................................................. 73
4.10 Graph after an edge contraction ......................................................................... 82

5.1 Vertices of the same connected components in continuous blocks ..................... 97
List of Tables

5.1 Software classes overview ........................................ 87
5.2 Mapping between brins (B), vertices (V), and edges (E) [32] ....... 90
5.3 Results for test set of grid graphs ................................ 100
5.4 Results for test set of r-outer planar graphs ...................... 101
5.5 Results for test set of series-parallel graphs ..................... 102
List of Algorithms

3.1 Finding maximum independent set with a nice path decomposition ........................................... 36
3.2 Calculating values for $D[i]$ and its descendents in a given nice tree decomposition .................. 41
3.3 Finding maximum independent set with a nice tree decomposition ......................................... 42
3.4 Calculating values for $D[i]$ and its descendents in a given rooted branch decomposition .......... 53
3.5 Finding maximum independent set with a rooted branch decomposition ............................... 53
4.1 The ST-Procedure ................................................................. 78
4.2 Computing the carving-width of a planar graph with binary search ....................................... 80
4.3 Constructing an optimal carving decomposition of a 2-connected planar graph ....................... 84
6.1 Solving hard problems on planar graphs ................................................................................. 104
Chapter 1

Introduction

Computational complexity identifies problems as efficiently solvable if there exists a polynomial time algorithm that solves it. However, many of the real-world problems do not fall in this category. Such problems are classified as the NP-HARD problems or worse. Fortunately, not all instances of NP-HARD problems are equally hard.

For example, the NP-COMPLETE problem INDEPENDENT-SET becomes very easy when the input graph is a tree. A simple two-dimensional array $D[]$ will help us find the answer. Let $D[u][b]$ be the maximum cardinality of independent sets for the tree rooted at $u$ with $u$ included in the set when $b = 1$, or with $u$ excluded from the set when $b = 0$. The cardinality of the maximum independent set for the tree will be $\max(D[root][0], D[root][1])$. A recursive dynamic programming formula to calculate $D$ can be described as follows.

if $i$ is a leaf, $D[i][b] = b$.

if $i$ is an internal node,

$$D[i][b] = \begin{cases} 
\sum_{j \text{ is a child of } i} \max(D[j][0], D[j][1]) & \text{if } b = 0, \\
1 + \sum_{j \text{ is a child of } i} D[j][0] & \text{if } b = 1.
\end{cases}$$
Using this formula, a simple tree traversal finds the cardinality of the maximum independent set in \( O(n) \) steps. By back-tracking, it is easy to construct the witness set as well.

In this thesis, only problems with input in the form of a graph are taken into consideration. In order to find efficient algorithms for them, we look into the parameterized versions of \textbf{NP-HARD} problems. In parameterized complexity, problems have an additional parameter to the input; this makes the running time a function of two variables: \( n \) - the instance size, and \( k \) - the additional parameter. A parameterized problem is Fixed-Parameter Tractable (FPT) if it has an algorithm with the running time in the form of \( O(f(k) \cdot p(n)) \) where \( f(k) \) can be any function, e.g. \( 2^k, 2^{k^2}, 2^{2^k}, \) etc. When the parameter \( k \) is small or is bounded by a reasonable constant, the problem can then be solved in polynomial time. Notice that for the parameterized version of \textbf{NP-HARD} problems, different parameters may lead to different complexity results.

\section*{1.1 Background}

A natural approach to solving hard problems is to try to use their solutions on trees to solve the same problems on tree-like graphs. The notions of tree decompositions and tree-width [26] were introduced in the early 1980s to determine the tree-likeness of a graph. A tree decomposition of a graph \( G \) is constructed by decomposing \( G \) into several "bags" consisting of vertices from \( G \) and building a tree on these bags.
The constructed tree also needs to satisfy some other conditions in order to be a tree decomposition of \( G \). Obviously, there are a lot of different tree decompositions for any given graph. Therefore, the width of a tree decomposition (maximum size of the bags on that tree decomposition subtracted by one) is used as a measure to rate tree decompositions. Since dynamic programming algorithms on tree decompositions for a given graph usually take time that is exponential in the width of the tree decompositions, a tree decomposition is better if it has smaller width. The tree-width of a graph \( G \) is defined as the smallest width among any tree decompositions of \( G \). By definition, the tree-width of a tree is 1. A graph is more “like” a tree if it has small tree-width.

Series-parallel graphs are the graphs that are most similar to trees. A series-parallel graph has a source vertex \( s \) and a sink vertex \( t \). Series-parallel graphs are defined recursively as follows.

1. A graph \( G \) consists of only two vertices \( s \) and \( t \) and an edge connecting them is a series-parallel graph.

2. A graph \( G \) is series-parallel if we can separate \( E(G) \) into two disjoint edge set \( E_1 \) and \( E_2 \) where their graphs \( G_i \) formed by \( E_i \) is a series-parallel graph with sources \( s_i \) and sink \( t_i \) for \( 1 \leq i \leq 2 \) such that:

   (a) either \( s_1 = s_2 \) and \( t_1 = t_2 \), we call \( G \) a parallel composition of \( G_1 \) and \( G_2 \);

   (b) or \( t_1 = s_2 \), we call \( G \) a series composition of \( G_1 \) and \( G_2 \).
In both cases, \( s_1 \) and \( t_2 \) are the source vertex and sink vertex of \( G \), respectively.

A constructive proof can be used to show that series-parallel graph has tree-width at most 2. In fact, a graph has tree-width at most 2 if and only if it is a sub-graph of a series-parallel graph [8]. Therefore, we can solve many hard problems on series-parallel graphs by slightly altering their solutions on trees.

Moreover, it has been shown by Courcelle [7] that many graph problems restricted to graphs of bounded tree-width are solvable in polynomial time. The polynomial time algorithms for these bounded tree-width problems are actually FPT algorithms for their parameterized version, where the input includes a tree decomposition for the given graph and the width of that decomposition is the parameter. Unfortunately, finding tree-width and a corresponding optimal tree decomposition is difficult, even for planar graphs. In fact, computing the tree-width of a graph is NP-COMPLETE [3]. Therefore, problems for graphs with bounded tree-width are not known to be efficiently solvable because we have to have a tree decomposition (which is hard to construct) ready at the beginning of their algorithms.

Theoretically, Bodlaender [5] gave an \( O(n) \) time algorithm for constructing a tree decomposition of width \( k \) if one exists. However, there is a huge constant factor hidden under the big-\( O \) notation. A thorough analysis in [28] gave the running time lower bound of \( O(2^{O(k^3)} \cdot n) \) for Bodlaender’s algorithm. This result suggests that Bodlaender’s constructive algorithm for tree-width and optimal tree decompositions cannot be used for most practical applications.
On the other hand, one can use a polynomial time approximation algorithm to construct a tree decomposition that has a reasonably small width, and then use this tree decomposition to solve the original problems. This tree-width approximation approach is still in FPT. However, the width of tree decompositions is the parameter $k$ for the parameterized problems and it normally is in the exponent of an exponential factor in $f(k)$. It follows that this approximated width can easily increase the running time of the whole process by a substantial amount.

This thesis looks into another type of graph decompositions: branch decompositions. Unlike tree decompositions, branch-width and optimal branch decompositions for planar graphs can be found in polynomial time. Finding the optimal branch decompositions in polynomial time enables us to solve problems for planar graphs with bounded branch-width efficiently. Therefore, an efficient implementation of the algorithm to find optimal branch decomposition is an important piece of a bigger puzzle.

1.2 Related Work

Even though branch-width and branch decompositions were introduced by Robertson and Seymour [26] a few years after they defined tree-width and tree decompositions in their Graph Minor series, branch-width is not as widely studied as tree-width. However, solving problems using optimal branch decompositions is a promising approach due to its fast constructive algorithm on planar graphs. More specifically,
Seymour and Thomas [29] gave an $O(n^2)$ algorithm, commonly referred to as the ST-PROCEDURE, for the problem of deciding whether a given planar graph has a branch decomposition of width at most $k$. This algorithm can be used to solve practical problems since there is no huge constant hidden under the big-$O$ notation.

In the same paper, Seymour and Thomas gave an $O(n^4)$ algorithm for constructing an optimal branch decomposition of a given planar graph based on the ST-PROCEDURE. Recently, the running time of that algorithm was improved to $O(n^3)$ by Gu and Tamaki [18] in 2005. A study comparing the two best known algorithms for optimal branch decompositions was conducted in 2007 [4]. In that study, the authors also provided several implementation improvements for both algorithms. However, ST-PROCEDURE is still the core function of all known exact algorithms for optimal branch decompositions for planar graphs.

1.3 Contributions of This Thesis

The main contributions of this thesis involve developing C++ classes that implement various functions related to branch and carving decompositions. These C++ classes were built on top of a light software package for building various types of graph decompositions by Dr. David Juedes that was developed in 2006. The original software was written in C++ with different classes for series parallel decompositions, tree decompositions, and nice tree decompositions. It also provided error checking functions to verify the correctness of series parallel decompositions, tree decompositions,
and nice tree decompositions. This software package was developed as a foundation for implementations of practical algorithms for graph-related problems in our research group.

We implemented the ST-PROCEDURE as the skeleton function for our branch decomposition and carving decomposition classes. The original $O(n^4)$ algorithm for constructing an optimal branch decomposition by Seymour and Thomas [29] was also included in this code. Finally, conversion procedures between branch decompositions, carving decompositions, and tree decompositions based on the constructive proof given by Robertson and Seymour [26] are implemented to prepare for future algorithms and improvements for the software package.

A secondary contribution of this thesis is the extensive explanation that we provide regarding branch decompositions, carving decompositions, medial and dual graphs, and the rat-catching game.

1.4 Thesis Overview

This thesis studies and discusses the implementation of an efficient algorithm to find optimal branch decompositions for planar graphs. The content is separated into five chapters. Chapter 2 provides fundamental concepts of NP-COMPLETE problems and fixed parameterized intractability. Chapter 3 discusses different types of graph decompositions, the relationships among the various types, and some algorithms that take advantage of those decompositions for general graphs. Chapter 4 reviews and
discusses the algorithms for finding branch-width and constructing optimal branch decompositions for planar graphs. Chapter 5 discusses an overview of the software, the implementation details, and some experimental results. Finally, chapter 6 concludes the thesis with potential applications of optimal branch decompositions for planar graphs for future work.
Chapter 2

NP-Complete Problems and Polynomial-time Algorithms

2.1 A Traditional Look at NP-Complete Problems

In classical complexity theory, the complexity class P denotes the class of all decision problems solvable in deterministic polynomial time and the complexity class NP denotes all the problems solvable in non-deterministic polynomial time.

An NP-COMPLETE problem is a problem in NP that is as hard as any other NP problems. Formally speaking, Problem A is NP-COMPLETE if $A \in \text{NP}$ and for any problem $B \in \text{NP}$, $B$ is polynomial time many-one reducible to $A$. It is trivial to see that P is a subset of NP. Some of the well-known NP problems are VERTEX-COVER, INDEPENDENT-SET, and DOMINATING-SET.

<table>
<thead>
<tr>
<th>VERTEX-COVER</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A graph $G = \langle V, E \rangle$ with $</td>
</tr>
<tr>
<td><strong>Output:</strong> The smallest $k$ such that there exists a set of vertices $V'$ of size $k$ and each edge in $E$ has at least one of its endpoints in $V'$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INDEPENDENT-SET</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A graph $G = \langle V, E \rangle$ with $</td>
</tr>
<tr>
<td><strong>Output:</strong> The largest $k$ such that there exists a set of vertices $V'$ of size $k$ and any two vertices $u$ and $v$ in $V'$ are not adjacent in $G$.</td>
</tr>
</tbody>
</table>
Dominating-Set

Input: A graph $G = (V, E)$ with $|V| = n$.
Output: The smallest $k$ such that there exists a set of vertices $V'$ of size $k$ and any vertex $u \in V$ is in $V'$ or adjacent to at least one vertex in $V'$.

These problems have been proved to be in NP-COMPLETE [17, 20].

However, the question of whether P equals to NP still remains open. The general hypothesis which most computer scientists believe is that P is a proper subset of NP. Under this hypothesis, the theory asserts that classical NP-COMPLETE problems such as Vertex-Cover, Independent-Set, or Dominating-Set do not have polynomial time algorithms.

2.2 Fixed Parameter Tractability and Polynomial-time Algorithms for NP-Complete Problems

Even though NP-COMPLETE problems are hard in general, several NP-COMPLETE problems are known to have polynomial time algorithms when restricted to inputs that have a special structure. Consider the following version of the Vertex-Cover problem:

Vertex-Cover

Input: A graph $G = (V, E)$ with $|V| = n$.
Parameter: An integer $k$.
Problem: Does there exist a set of vertices $V'$ of size $k$ such that each edge in $E$ has at least one of its endpoints in $V'$?

This problem is a slight variant of the decision problem for Vertex-Cover. It can be solved in time $O(n)$ if the parameter $k$ is fixed or bounded by a constant [13],
i.e., if we want to find whether a “big” graph has a “small” vertex cover, we can do so in $O(n)$ steps. More precisely, the best known algorithm for VERTEX-COVER has running-time $O(1.27^k \cdot n)$ [6]. Therefore, this algorithm runs in linear time when $k$ is bounded by a constant, because the factor $1.27^k$ becomes a constant and can be hidden under the big-$O$ notation.

The linear time algorithm for VERTEX-COVER shows that VERTEX-COVER is in the class FPT that contains all fixed-parameter tractable problems. A problem is considered fixed-parameter tractable if it can be solved in time $O(f(k) \cdot p(n))$ where $n$ is the instance size and $k$ is the additional parameter [13]. The class FPT is a foundation class in a hierarchy of complexity classes based on parameterized complexity. Let us make an assumption that the parameterized version does not require any additional information; or if it does, the additional information can be constructed in polynomial time. When the parameter $k$ is bounded by a constant $C$, we can solve the original problem in polynomial time by trying each and every value of $k$ from 0 to $C$ and stopping when the parameterized problem returns YES. This assumption is a critical requirement on how fast the original version can be solved because exponential pre-processing time negates possibility of polynomial solution.

Choosing the parameter $k$ plays an important role on whether a problem is in FPT or not. For example, INDEPENDENT-SET is not known to be in FPT if we use the same parameter from VERTEX-COVER, i.e., the size of the independent set. In fact, this parameterized version of INDEPENDENT-SET is complete for $W[1]$, a class in
the $W$ hierarchy of fixed-parameter intractable classes which will be discussed in the next section. On the other hand, INDEPENDENT-SET is in FPT if we include a tree decomposition in the input and choose $k$ to be the width of that tree decomposition. We will discuss graph decompositions and how we use them to solve NP-COMPLETE problems in polynomial time later in chapter 4.

2.3 Fixed Parameter Intractability and the $W$ Hierarchy

The previous section mentions that a parameterized problem is tractable if it can be solved in $O(f(k) \cdot p(n))$ time. What if a parameterized problem cannot be solved in $FPT$ time like in the case of INDEPENDENT-SET and DOMINATING-SET? Are they all equally hard in the FPT sense or is there a way to separate them into different complexity classes? The answer to these questions led to the weft hierarchy (known as the W-hierarchy) that was introduced by Downey and Fellows [10, 11] for fixed-parameter intractable problems. Since the W-hierarchy was built based on the two-dimensional complexity, we need to have a reduction in this domain in order to prove hardness and completeness of parameterized problems. Such reduction is called the FPT reduction.
Definition 2.1 ([24]). A parameterized problem $Q$ is FPT-reducible to a parameterized problem $Q'$ if there is a function $f$ and an algorithm $A$ that converts any instance $(x, k)$ of $Q$ to an instance $(x', k')$ of $Q'$ such that:

(i) Algorithm $A$ runs in time $f(k) \cdot |(x,k)|^c$ for some constant $c$, and

(ii) $(x, k) \in Q \iff (x', k') \in Q'$

In order to build the W-hierarchy, Downey and Fellows [13] defined the parameterized version of CIRCUIT-SAT as follows:

<table>
<thead>
<tr>
<th>p-CIRCUIT-SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A circuit $C$ of size $n$.</td>
</tr>
<tr>
<td><strong>Parameter:</strong> An integer $k$.</td>
</tr>
<tr>
<td><strong>Problem:</strong> Does there exist a truth assignment of weight $k$ that evaluates $C$ to true?</td>
</tr>
</tbody>
</table>

For this problem, the weight of a truth assignment is the number of input gates assigned to TRUE. A circuit has exactly one output gate and its gates are categorized into two types: small gates and large gates. Small gates are gates with fan-in 2 while large gates are gates with arbitrary fan-in (greater than 2). Notice that the boundary value 2 decides whether a given gate is small or large. Replacing this boundary value with any fixed constant will not significantly change any of the following results. From this categorization, the “weft” of a circuit with depth bounded by a constant is defined as follows.

Definition 2.2. The “weft” of a circuit with depth bounded by a constant is the maximum number of large gates on any path from an input gate to the output gate.
Using “weft” to separate circuit into classes, we have the following parameterized problem from [13].

<table>
<thead>
<tr>
<th>Weft-t-Depth-h-Circuit-Sat</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A circuit C of size n with weft t and depth h where h ≥ t.</td>
</tr>
<tr>
<td><strong>Parameter:</strong> An integer k.</td>
</tr>
<tr>
<td><strong>Problem:</strong> Does there exist a truth assignment of weight k that causes C to evaluate to true?</td>
</tr>
</tbody>
</table>

Each class \( W[t] \) in the W-hierarchy is defined as follows.

**Definition 2.3.** A problem is in \( W[t] \) if it is FPT-reducible to \( \text{Weft}-t-\text{Depth}-h-\text{Circuit-Sat} \) for some h.

**Definition 2.4.** A problem is in \( W[P] \) if it is FPT-reducible to \( p-\text{Circuit-Sat} \).

Consequently, the relationships between the classes \( FPT, W[t], \) and \( W[P] \) are:

\[
FPT \subseteq W[1] \subseteq W[2] \subseteq ..W[t] \subseteq .. \subseteq W[P]
\]

The following result is not new, but we give the complete proof for this theorem as an explanation for those who are not quite familiar with the W-hierarchy.

**Theorem 2.5.** \( \text{INDEPENDENT-Set} \) is in \( W[1] \). \( \text{DOMINATING-Set} \) is in \( W[2] \).

**Proof.** In order to prove this result, we only need to convert the input graphs to constant-depth circuits of weft one and two, respectively. Let \( G = \langle V, E \rangle \) be the input graph for \( \text{INDEPENDENT-Set} \) or \( \text{DOMINATING-Set} \). We will build a circuit \( C \) of depth two for \( G \). In both cases, each vertex \( u \) in \( V \) corresponds to one and only
one input gate $g_u$ of $C$. The output gate is a large AND gate with inputs from all gates at depth one. A weight-$k$ assignment for $C$ corresponds to a vertex set $V'$ of $G$ where $V'$ contains $u$ if $g_u$ is set to true in the assignment.

![Diagram](image)

Figure 2.1: Converting a graph to a circuit for Independent-Set

For **Independent-Set**, the input gates are connected directly to NOT gates (figure 2.1). For each edge $(u, v)$ in $E$, we have an AND gate at depth one that takes inputs from the NOT gates of $g_u$ and $g_v$. In figure 2.1, the circuit on the right is created for the graph on the left to solve **Independent-Set**. Large gates are ellipses while small gates are circles.

We show that a truth assignment that causes $C$ to evaluate to true is an independent set of $G$. If it is not, let $u$ and $v$ be two vertices in $V'$ (the vertex set corresponds to the given truth assignment) that violate the independence condition. Since $u$ and $v$ are connected, there must be an AND gate $g_e$ in $C$ with inputs from the NOT
gates of \( g_u \) and \( g_v \). This gate \( g_e \) evaluates to FALSE because both of its inputs evaluate to FALSE. Hence, the output gate evaluates to FALSE which contradicts our assumption. Therefore, \( V' \) is an independent set. Similarly, an independent set \( V' \) of \( G \) corresponds to a truth assignment that causes \( C \) to evaluate to true. Moreover, \( C \) has weft 1 because it only has one large gate which is the output gate. It follows that \textsc{Independent-Set} is FPT-reducible to \textsc{weft-1-depth-3-Circuit-Sat}, i.e., \textsc{Independent-Set} is in \( W[1] \).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Converting a graph to a circuit for Dominating-Set}
\end{figure}

For \textsc{Dominating-Set}, input gates are normal gates. For each vertex \( u \) in \( V \) with a neighbor set \( V_u \) consisting of \( u \) and all vertices adjacent to \( u \), we have an OR gate \( g'_u \) at depth one that takes inputs from all input gates \( g_v \) where \( v \in V_u \).

A truth assignment that causes \( C \) to evaluate to true is a dominating set of \( G \). If it is not, let \( u \) be a vertex in \( V \) that is not dominated. Since \( u \) is not dominated, none of the vertices in \( V_u \) is in the dominating set, i.e., \( g_v \) is FALSE for all \( v \in \).
Consequently, \( g'_a \) evaluates to FALSE. Hence, the output gate also evaluates to FALSE. This contradicts our assumption. Similarly, a dominating set \( V' \) of \( G \) corresponds to a truth assignment that evaluates \( C \) to true. Finally, we want to justify the weft of \( C \). The output gate of \( C \) is a large gate. Any gate at depth one as described above can have at least one input and at most \( n \) inputs. So they are also large gates. It follows that \( C \) has weft two. Therefore, \textsc{Dominating-Set} is FPT-reducible to \textsc{weft-2-depth-2-Circuit-Sat}, i.e., \textsc{Dominating-Set} is in \( W[2] \).

The circuit weft in our proof for \textsc{Dominating-Set} depends on the degree of each vertex in \( G \). If the input graph has bounded degree, gates at depth one become small gates. Hence, the weft of our circuit will be decreased to one. We have the following corollary.

**Corollary 2.6.** The \textsc{Dominating-Set} problem restricted to graphs of degree bounded by \( c \), for each constant \( c \), is in \( W[1] \).

Not only are \textsc{Independent-Set} and \textsc{Dominating-Set} in \( W[1] \) and \( W[2] \) respectively, they are also the hardest problems in these complexity classes. In other words, \textsc{Independent-Set} and \textsc{Dominating-Set} are \( W[1] \)-Complete and \( W[2] \)-Complete. We give the following proof sketch to help readers get more familiar with the W-hierarchy.

Proof sketch. We knew that Independent-Set and Dominating-Set are in $W[1]$ and $W[2]$ from Theorem 2.5. Therefore, we only need to prove that they are hard for $W[1]$ and $W[2]$.

Downey and Fellows [11] proved that Independent-Set is $W[1]$-Hard by providing an FPT reduction from weft-$1$ anti-monotone circuit with fan-in bounded by two to Independent-Set. Moreover, $W[1, s]$ is defined in [11] as the class of problems that are FPT-reducible to weft-$1$ anti-monotone circuit with fan-in bounded by $s$. It follows that Independent-Set is hard for $W[1, 2]$. However, it is proven in [11] that the class $W[1]$ collapsed to $W[1, 2]$. Therefore, Independent-Set is also hard for $W[1]$.


```
Weighted-CNF-Satisfiability

Input: A boolean formula $F$ in conjunctive normal form.
Parameter: An integer $k$.
Problem: Does there exist a truth assignment of weight $k$ that evaluates $F$ to true?
```

It is easy to see that Weighted-CNF-Satisfiability is $W[2]$-Complete. Hence, Dominating-Set is $W[2]$-Hard. \hfill \Box
On a side note, Planar-Dominating-Set is in FPT. Using the bounded search tree technique, Downey and Fellows [12] introduced an FPT algorithm that solves Planar-Dominating-Set in $O(11^k|G|)$. Indeed, this problem can be solved in $O(2^{O(\sqrt{k})} \cdot p(n))$ steps using more advanced techniques which take advantage of the fact that a planar graph has tree width at most $c \cdot \sqrt{k}$ for some constant $c$ if it has a dominating set of size $k$ [2, 19, 16]. The best known value of $c$ is 9.546 due to Fomin and Thilikos [16].
CHAPTER 3

Graph Decompositions and FPT Algorithms

In this chapter, we discuss several different types of graph decompositions: path decompositions, tree decompositions, and branch decompositions. For the purpose of this thesis, a decomposition of graph $G = (V, E)$ is a graph $X$ with a simple structure where each node of $X$ corresponds to a subset of $V$ or $E$ and satisfies some specific conditions regarding $G$. For tree decompositions, the graph $X$ is in the form of a tree and each node is a bag which corresponds to a subset of $V$. A path decomposition is essentially a special case of a tree decomposition where $X$ is a path. In the case of branch decompositions, $X$ is also a tree where each internal node has degree exactly three and each leaf node corresponds to one and only one edge in $E$. There are also other conditions that need to be satisfied for each type of decomposition.

The width of a decomposition is normally defined as the maximum cardinality of the nodes in $X$. Finally, the tree-width, path-width, or branch-width of a graph is the minimum width of any tree decomposition, path decomposition, or branch decomposition for that graph, respectively. For historical reasons, the width for tree decompositions or path decompositions is calculated by subtracting one from the
maximum cardinality of the nodes in $X$. The reason behind this is to guarantee that tree-width for a tree is one and the path-width for a path is one.

This chapter also explores efficient dynamic-programming algorithms to solve \textsc{Independent-Set}, a well-known \textsc{NP-Complete} problem, when given a decomposition with bounded width. These algorithms are in \textsc{FPT} with the running time of $O(2^{O^*(k)} \cdot p(n))$ where $k$ is the width of the given decomposition and $n$ is the size of the graph. Even though finding an optimal decomposition for general graphs is hard [3], these algorithms come in handy when we have either an optimal decomposition for special types of graphs or a decomposition that is nearly optimal, i.e., some approximately optimal solution. More specifically, graph problems for planar graphs can be solved efficiently by constructing optimal branch decompositions in polynomial time and applying efficient dynamic-programming algorithms that takes as input the original graph and the optimal branch decomposition.

\subsection{Path Decompositions}

Path decomposition was introduced by Robertson and Seymour in the first paper of their Graph Minor series [25].
3.1.1 Definitions

Definition 3.1 ([25]). Given a graph $G = \langle V, E \rangle$. A path decomposition of $G$ is a sequence $X_1, X_2, ..., X_r$ of subsets of $V$ that satisfies the following conditions.

(i) For every edge $e = \langle u, v \rangle \in E$, there exists an $i$ where $1 \leq i \leq r$ such that 
\[ \{u, v\} \subseteq X_i \]

(ii) For all $i, j, $ and $k$ such that $1 \leq i \leq j \leq k \leq r$, 
\[ X_i \cap X_k \subseteq X_j. \]

The width of a path decomposition is $\max_{1 \leq i \leq r} \{|X_i|\} - 1$. The path-width of a graph $G$ is the minimum width of any path decompositions of $G$.

In order to apply dynamic-programming on path decompositions easily, we introduce an alternate definition of a nice path decomposition as follows.

Definition 3.2 (e.g. [5]). Given a graph $G = \langle V, E \rangle$. A nice path decomposition of $G$ is a path decomposition $X_1, X_2, ..., X_r$ with the following additional conditions.

(i) $X_1 = X_r = \emptyset$

(ii) $X_i$ ($2 \leq i \leq r$) is either an introduce node or a forget node.

- $X_i$ is an introduce node if $\exists u \in V$ such that $X_i = X_{i-1} \cup \{u\}$
- $X_i$ is a forget node if $\exists u \in X_{i-1}$ such that $X_i = X_{i-1} \setminus \{u\}$

Note that the notation $A \setminus B$ used above denotes the set minus operation, i.e., this is the set of all elements in $A$ and not in $B$. Sometimes, this set is denoted by $A - B$. 

Note also that in a nice path decomposition, each vertex \( u \in V \) is added and removed exactly once. Hence, there is one node that introduces \( u \) and one other node that forgets \( u \) in a nice path decomposition. Therefore, the number of nodes in a nice path decomposition is exactly \( 2 \cdot |V| + 1 \).

We also define the sub-graph \( G[X_i] \) of \( G \) regarding a path decomposition \( \{X_i\}_1^r \) as a graph \( \langle V_i, E_i \rangle \) with \( V_i = \bigcup_{k=1}^{i} X_k \) and \( E_i = \{\langle u, v \rangle \mid \{u, v\} \subseteq V_i \text{ and } \langle u, v \rangle \in E(G)\} \).

### 3.1.2 Solving Independent-Set with Path Decompositions

We use dynamic-programming on subsets to solve \textsc{ Independent-Set } given a graph \( G = \langle V, E \rangle \) and a nice path decomposition \( \{X_i\}_1^r \). The dynamic-programming table is defined as a two-dimensional array \( D[i, S] \) where \( 1 \leq i \leq r \) and \( S \subseteq X_i \). \( D[i, S] \) is the maximum size of all independent sets \( I \) of \( G[X_i] \) such that \( I \cap X_i = S \).

Notice that \( G[X_r] = G \) and \( X_r = \emptyset \). Therefore, \( D[r, \emptyset] \) is the size of the maximum independent set of \( G \). Hence, all we need to do is to calculate the values in the dynamic-programming table \( D \) and the answer is \( D[r, \emptyset] \). Moreover, since \( X_1 \) is the empty set, \( D[1, \emptyset] = 0 \).

For any \( i \) such that \( 2 \leq i \leq r \), we have two cases.
If $X_i$ is an introduce node. Let $u = X_i \setminus X_{i-1}$.

$$D[i, S] = \begin{cases} 
D[i - 1, S] & \text{if } u \notin S \\
D[i - 1, S \setminus \{u\}] + 1 & \text{if } u \in S \text{ and } S \text{ is an INDEPENDENT-SET of } G[X_i] \\
-\infty & \text{otherwise}
\end{cases}$$

We verify the correctness of the above formula. In the third case, since $S$ is not an independent set itself, there can not be any independent set that has $S$ as a subset. So we assign a value of $-\infty$ to $D[i, S]$.

In the other cases, let $I$ be the independent set corresponding to $D[i, S]$. We have $I \cap X_i = S$.

In the first case, $I \cap X_{i-1} = S$ since $u \notin S \subset I$. Therefore, $|I| \leq D[i - 1, S]$. In fact, $|I|$ must be equal to $D[i - 1, S]$. Otherwise, if $|I| < D[i - 1, S]$, let $I'$ be the independent set of $G[X_{i-1}]$ corresponding to $D[i - 1, S]$. Since $I' \cap X_i = S$ and $I'$ is also an independent set of $G[X_i]$, we have $|I'| \leq D[i, S] = |I|$. It follows that we have both $|I'| \leq |I|$ and $|I'| > |I|$. Therefore, the assumption is not true. Hence, $|I| = D[i - 1, S]$.

Similarly, we can calculate $D[i, S]$ in the second case by using the value of $D[i - 1, S \setminus \{u\}] + 1$.

If $X_i$ is a forget node. Let $u = X_{i-1} \setminus X_i$.

$$D[i, S] = \max(D[i - 1, S], D[i - 1, S \cup \{u\}])$$
This formula gives us the correct value of $D[i, S]$. Otherwise, assume that the actual value of $D[i, S] > \max(D[i - 1, S], D[i - 1, S \cup \{u\})$. Let $I$ be an independent set of $G[X_i]$ such that $I \cap X_i = S$ and $|I| > \max(D[i - 1, S], D[i - 1, S \cup \{u\})$. Since $X_{i-1} = X_i \cup \{u\}$, it follows that $I \cap X_{i-1} = S$. Therefore, $I$ is also an independent set in $G[X_{i-1}]$. That means, $D[i - 1, S] \geq |I| > \max(D[i - 1, S], D[i - 1, S \cup \{u\}) > D[i - 1, S]$. This is always false, so the assumption is wrong. That means the formula is correct.

Putting it all together, we have the following path-width based algorithm for \textsc{Independent-Set}.

### 3.1.3 Running Time Analysis

From the pseudo-code of algorithm 3.1, we can see that the algorithm runs in time $O((r - 1) \cdot k^2 \cdot 2^{k+1})$ where $k$ is the width of the given path decomposition. The factor of $k^2$ comes from the step where we need to verify if $S$ is an independent set. However, we only need to check if $u$ is connected to any vertex in $S$. If it is, $S$ is not an independent set. Otherwise, we can directly assign $D[i - 1, S \setminus u] + 1$ to $D[i, S]$. The reason behind this logic is that all the values of $D[i - 1]$ were already computed at that point. $S \setminus \{u\}$ is not an independent set because $S$ is not an independent set with $u$ being an independent vertex in $S$. That means $D[i - 1, S \setminus \{u\}]$ is $-\infty$ in the previous loop. Hence, $D[i, S]$ is also $-\infty$. Therefore, with a good implementation, the procedure will run in time $O((r - 1) \cdot k \cdot 2^{k+1})$. 
Algorithm 3.1 Finding maximum independent set with a nice path decomposition

Require: A graph $G$ and a nice path decomposition $X$
Ensure: The cardinality of maximum independent sets of $G$.

1: initialize all elements of $D$ to $-\infty$
2: $D[1, \emptyset] \leftarrow 0$
3: for $i$ from 2 to $r$ do
4: if $X[i]$ is an introduce node then
5: $u \leftarrow X[i] \setminus X[i-1]$
6: for all $S \subseteq X[i]$ do
7: if $u \notin S$ then
8: $D[i, S] \leftarrow D[i - 1, S]$
9: else if $S$ is an independent-set then
10: $D[i, S] \leftarrow D[i - 1, S \setminus \{u\}] + 1$
11: else
12: $D[i, S] \leftarrow -\infty$
13: end if
14: end for
15: else
16: $u \leftarrow X[i - 1] \setminus X[i]$
17: for all subset $S$ of $X[i]$ do
18: $D[i, S] \leftarrow \max(D[i - 1, S], D[i - 1, S \cup \{u\}])$
19: end for
20: end if
21: end for
22: return $D[r, \emptyset]$

Since $X$ is a nice path decomposition, $r = 2n + 1$. The total running time is $O(n \cdot k \cdot 2^k)$.

3.2 Tree Decompositions

A tree decomposition is the generalization of a path decomposition. First introduced by Robertson and Seymour [26] in 1984, tree decompositions have been widely explored and studied in graph theory research.
3.2.1 Definitions

Definition 3.3 ([26]). Given a graph $G = (V, E)$. A tree decomposition of $G$ is a pair $(T, X)$ where $T = (I, f)$ is a tree and $X$ is a mapping from $I$ to the power set of $V$ that satisfies the following conditions.

(i) $\bigcup_{i \in I} X_i = V$

(ii) For every edge $e = (u, v) \in E$, there exists an $i \in I$ such that $\{u, v\} \subseteq X_i$

(iii) For all $u, v, w \in I$ that $w$ is along the path from $u$ and $v$ in $T$, $X_u \cap X_v \subseteq X_w$

In a fashion similar to the definition of a path decomposition, the width of a tree decomposition is $\max_{i \in I} \{|X_i|\} - 1$. The tree-width of a graph $G$ is the minimum width of any tree decompositions of $G$.

For ease of use, we give an alternate definition of a nice tree decomposition as follows.

Definition 3.4 (e.g. [5]). Given a graph $G = (V, E)$. A nice tree decomposition of $G$ is a tree decomposition $(T, X)$ with the following additional conditions:

(i) $T = (I, f)$ is a binary tree.

(ii) If $i \in I$ is a leaf, $X_i$ is an empty set.

(iii) The root node is also an empty set.

(iv) If $i \in I$ is an internal node, $X_i$ can be either a join node, an introduce node, or a forget node.
• $X_i$ is a join node if it has exactly two children $X_j$ and $X_k$ and $X_i = X_j = X_k$

• $X_i$ is an introduce node if it has exactly one child $X_j$, and there exists $u \in X_i$ such that $X_i = X_j \cup \{u\}$

• $X_i$ is a forget node if it has exactly one child $X_j$, and there exists $u \in X_j$ such that $X_i = X_j \setminus \{u\}$

Let $T_i$ be the sub-tree rooted at $i$ on $T$. The sub-graph $G[X_i]$ of $G$ regarding a tree decomposition $\{X_i\}_1^i$ is defined as a graph $\langle V_i, E_i \rangle$ with $V_i = \bigcup_{X_k \in T_i} X_k$ and $E_i = \{ \langle u, v \rangle \mid \{u, v\} \subseteq V_i \text{ and } \langle u, v \rangle \in E(G) \}$.

3.2.2 Solving Independent-Set with Tree Decompositions

Dynamic-programming on subset approach can be used to solve INDEPENDENT-SET given a graph $G = \langle V, E \rangle$ and a nice tree decomposition $\{T, X\}$. It is proved that given an arbitrary tree decomposition, one can construct a nice tree decomposition of the same width with at most $4 \cdot |V|$ nodes in polynomial time [21]. Therefore, we can assume that the given tree decomposition has $O(|V|)$ nodes. Without loss of generality, we can assume that $X_0$ is the root of $T$. Similar to the algorithm with nice path decompositions, the dynamic-programming table is defined as a two-dimensional array $D[i, S]$ where $i \in I$ and $S \subseteq X_i$. $D[i, S]$ is the maximum size of all independent sets $I$ of $G[X_i]$ such that $I \cap X_i = S$. Notice that $G[X_0] = G$ and $X_0 = \emptyset$. Therefore, $D[0, \emptyset]$ is the size of the maximum independent set of $G$. Hence, all we need to do is
to calculate the values the dynamic-programming table $D$ and the answer is $D[0, \emptyset]$.

Moreover, since $X_i$ is an empty set for each leaf $i$, $D[i, \emptyset] = 0$.

For any internal node $i$, we have three cases.

**If $X_i$ is an introduce node.** Let $X_j$ be $X_i$’s child and $u = X_i \setminus X_j$.

$$D[i, S] = \begin{cases} 
D[j, S] & \text{if } u \notin S \\
D[j, S \setminus \{u\}] + 1 & \text{if } u \in S \text{ and } S \text{ is an Independent-Set in } G[X_i] \\
-\infty & \text{if } u \in S \text{ and } S \text{ is not an Independent-Set in } G[X_i]
\end{cases}$$

In this case, the introduce node works exactly the same as an introduce node in a nice path decomposition. Therefore, the correctness of this formula follows directly from the one for nice path decompositions.

**If $X_i$ is a forget node.** Let $X_j$ be $X_i$’s child and $u = X_j \setminus X_i$.

$$D[i, S] = \max(D[j, S], D[j, S \cup \{u\}])$$

Same as the case of an introduce node, a forget node in a nice tree decomposition works exactly the same as a forget node in a nice path decomposition.

**If $X_i$ is a join node.** Let $X_j$ and $X_k$ be $X_i$’s children.


We show that the above formula for a join node is correct. From the definition of a join node, $X_j = X_k = X_i$. Moreover, it is trivial to see that $G[X_k]$ and $G[X_j]$ are sub-graphs of $G[X_i]$. 
Notice that if $S$ is not an independent set of $G[X_i]$, the formula above will give us $D[i, S] = -\infty$ since $D[j, S]$ and $D[k, S]$ were assigned to $-\infty$ in the previous steps. So we can safely assume that $S$ is an independent set of $G[X_i]$.

Let $I$ be an independent set of $G[X_i]$ with maximum size such that $I \cap X_i = S$. We have $|I| = D[i, S]$ by definition.

We prove that $|I| = D[j, S] + D[k, S] - |S|$.

Let $I_j = I \cap V(G_j)$ and $I_k = I \cap V(G_k)$. It follows that $I = I_j \cup I_k$. Since $I$ is an independent set in $G[X_i]$, $I_j$ and $I_k$ are also independent set in $G[X_j]$ and $G[X_k]$, respectively. Moreover, $I_j \cap X_j = I_j \cap X_i = S$ and $I_k \cap X_k = I_k \cap X_i = S$. $I_j \cap I_k \subseteq X_i$ by the third condition of a tree decomposition. Therefore, $I_j \cap I_k = S$. Hence, $|I| = |I_j| + |I_k| - |S|$

On the other hand, $|I_j| = D[j, S]$. This is true because if $|I_j| < D[j, S]$, assume that there is another independent set $I'_j$ of $G[X_j]$ where $I'_j \cap X_j = S$ and $|I'_j| = D[j, S]$ such that $|I'_j| > |I_j|$. Then, $I'_j \cap I_i = S$ and $I'_j \cap I_k$ must be $S$ by the third condition of a tree decomposition. Since $S$ is an independent set, $I' = I'_j \cup I_k$ is also an independent set for $G[X_i]$. Moreover, $I' \cap X_i = S$ and $|I'| = |I'_j| + |I_k| - |S| > |I_j| + |I_k| - |S| = D[i, S]$. This contradicts with the definition of $D[i, S]$. Hence, there is no such $I'_j$.


Putting the cases together, we have algorithm 3.2 to calculate each row $i$ of the dynamic programming table $D$. 
Algorithm 3.2 Calculating values for $D[i]$ and its descendents in a given nice tree decomposition

**Require:** A graph $G$, a nice tree decomposition $X$, an integer $i$.

**Ensure:** All elements $D[i]$ and its descendents in $X$ are calculated.

1: initialize $D[i]$ to $-\infty$
2: if $X[i]$ is a leaf then
3:   $D[i] \leftarrow 0$
4: else
5:   let $X[j]$ be $X[i]$’s first child
6:   run algorithm 3.2 recursively on $(G, X, j)$
7: if $X[i]$ is an introduce node then
8:   $u \leftarrow X[i] \setminus X[j]$
9:   for all $S \subset X[i]$ do
10:      if $u \notin S$ then
11:         $D[i, S] \leftarrow D[j, S]$
12:      else if $S$ is an independent-set then
13:         $D[i, S] \leftarrow D[j, S \setminus \{u\}] + 1$
14:      else
15:         $D[i, S] \leftarrow -\infty$
16:   end if
17: end for
18: else if $X[i]$ is a forget node then
19:   $u \leftarrow X[j] \setminus X[i]$
20:   for all $S \subset X[i]$ do
21:      $D[i, S] \leftarrow \max(D[j, S], D[j, S \cup \{u\}])$
22:   end for
23: else
24:   let $X[k]$ be $X[i]$’s second child
25:   run algorithm 3.2 recursively on $(G, X, k)$
26: for all $S \subseteq X[i]$ do
27:   $D[i, S] \leftarrow D[j, S] + D[k, S] - |S|$  
28: end for
29: end if
30: end if

Finally, algorithm 3.3 uses algorithm 3.2 as a sub-routine to compute the cardinality of maximum independent sets for a graph $G$ given a tree decomposition of $G$. 
Algorithm 3.3 Finding maximum independent set with a nice tree decomposition

Require: A graph $G$ and a nice tree decomposition $X$ rooted at $X[0]$
Ensure: The cardinality of maximum independent sets of $G$.
1: run algorithm 3.2 on $(G, X, 0)$
2: return $D[0, \emptyset]$

3.2.3 Running Time Analysis

From the pseudo-code of algorithms 3.2 and 3.3, we can see that algorithm 3.3 runs in time $O(m \cdot k^2 \cdot 2^{k+1})$ where $k$ is the width of the given tree decomposition and $m$ is the number of nodes in the tree decomposition. Similar to the analysis of algorithm 3.1, the factor $k^2$ comes from the step where we need to check if $S$ is an independent set in the case of an introduce node. We can apply the same technique to reduce the running time of this step from $k^2$ to $k$. Therefore, the total running time will be $O(m \cdot k \cdot 2^k)$.

Since $m = O(|V|)$, the total running time is $O(n \cdot k \cdot 2^k)$.

3.3 Branch Decompositions

Robertson and Seymour defined the notion of a branch decomposition of a graph and the related notion of branch-width [27] a few years after tree-width was introduced. Unlike tree-width, it is known that there is a polynomial time algorithm to construct an optimal branch decomposition for planar graphs. Moreover, we can convert a branch decomposition to a tree decomposition with reasonable width and vice versa. This section introduces some basic concepts about branch decompositions.
for general graphs and discusses an example of a branch-width based algorithm for
\textsc{Independent-Set}. The constructive algorithm to produce optimal branch decom-
positions for planar graphs will be discussed in detail in the next chapter.

### 3.3.1 Definitions

**Definition 3.5 ([27]).** Given a graph $G = \langle V, E \rangle$, a branch decomposition of $G$ is a
pair $\langle T, X \rangle$ where $T$ is a tree with each internal node having degree exactly three and $X$ is a bijective mapping from the leaves of $T$ to $E$.

Notice that the number of leaves in a branch decomposition is $|E|$. Because of the
fixed structure of a branch decomposition, the number of internal nodes in the tree
is exactly $|E| - 2$.

Given a branch decomposition $\langle T, X \rangle$, the midset of each edge $f$ in $T$ is defined
as follows.

**Definition 3.6 ([27]).** Let $T_1$ and $T_2$ be the two components of $T \setminus \{f\}$. The midset
of $f$ is the set of all vertices $v$ in $V$ such that there exist two leaves $l_1 \in T_1$ and $l_2 \in T_2$
and $v$ is incident to both $X(l_1)$ and $X(l_2)$.

An example of midsets and width of branch decompositions is demonstrated in
figure 3.1. A branch decomposition is given for the graph on the left. The midsets
are written in blue next to the edges.
Figure 3.1: A branch decomposition of width three

**Definition 3.7.** Let $\langle T, X \rangle$ be a branch decomposition of $G$. The width of $\langle T, X \rangle$ is

$$\max_{f \in T} \{|\text{midset}(f)|\}.$$ 

Similar to tree-width, the *branch-width* of a graph $G$ is the minimum width of any branch decompositions of $G$.

From the above definitions, we have the following properties of midsets.

**Lemma 3.8.** Let $f_1$, $f_2$, and $f_3$ be three edges incident to the same vertex $t$ in $T$.

Let $M_1 = \text{midset}(f_1)$, $M_2 = \text{midset}(f_2)$, $M_3 = \text{midset}(f_3)$, and $M = M_1 \cup M_2 \cup M_3$.

Any vertex $v$ in $M$ must belong to at least two of the three sets $M_1$, $M_2$, and $M_3$. 
Figure 3.2: Midsets of three edges incident to the same vertex on a branch decomposition

Proof. W.L.O.G., we can assume that $M_1$ is not empty. Let $u$ be a vertex in $M_1$. We will prove that $u$ is also in $M_2 \cup M_3$. Since $f_1$, $f_2$, and $f_3$ are incident to the same vertex $t$ of $T$; they separate $T$ into three components $T_1$, $T_2$, and $T_3$. Let $V_i$ be a vertex set of $V$ that are the union of all the leaves in $T_i$ with $1 \leq i \leq 3$. From the definition of midset, we have $M_1 = V_1 \cap (V_2 \cup V_3)$, $M_2 = V_2 \cap (V_3 \cup V_1)$, and $M_3 = V_3 \cap (V_1 \cup V_2)$.

Since $u \in M_1 = \text{midset}(f_1)$, we have $u \in V_1$ and $u \in V_2 \cup V_3$.

If $u \in V_2$: $u \in V_2 \cap V_1 \subseteq V_2 \cap (V_1 \cup V_3) = M_2$.

If $u \in V_3$: $u \in V_3 \cap V_1 \subseteq V_3 \cap (V_1 \cup V_2) = M_3$.

Therefore, $u \in M_2 \cup M_3$. 

□
Corollary 3.9. Let $f_1$, $f_2$, and $f_3$ be three edges incident to the same vertex $t$ in $T$. Let $M_1 = \text{midset}(f_1)$, $M_2 = \text{midset}(f_2)$, $M_3 = \text{midset}(f_3)$, and $M = M_1 \cup M_2 \cup M_3$. The edges $f_1$, $f_2$, and $f_3$ separate the branch decomposition tree into three sub-trees $T_1$, $T_2$, and $T_3$, respectively. Let $V_i$ be the vertex subset of $V$ that contains the union of all leaves on $T_i$ with $1 \leq i \leq 3$. Any vertex $v$ in $M$ must belong to at least one of the two sets $V_i$ and $V_j$ for all $1 \leq i < j \leq 3$.

Proof. We prove the claim for $i = 1$ and $j = 2$. Let $u$ be a vertex in $M$. From the proof of lemma 3.8, if $u \in M_1$, $u \in V_1$. Similarly, if $u \in M_2$, $u \in V_2$. If $u \in M_3$, $u \in V_1 \cup V_2$. Hence $u$ is in at least one of the two sets $V_1$ and $V_2$. \qed

Lemma 3.10. Let $f_1$, $f_2$, and $f_3$ be three edges incident to the same vertex $t$ in $T$. Let $M_1 = \text{midset}(f_1)$, $M_2 = \text{midset}(f_2)$, $M_3 = \text{midset}(f_3)$. Let $X_1$, $X_2$, $X_3$, and $X_4$ be defined as follows: $X_1 = M_1 \setminus M_2$, $X_2 = M_2 \setminus M_3$, $X_3 = M_3 \setminus M_1$, and $X_4 = M_1 \cap M_2 \cap M_3$. We have the following relationship among $M_1, M_2, M_3, X_1, X_2, X_3,$ and $X_4$:

(i) $X_i \cap X_j = \emptyset$ for all $1 \leq i < j \leq 4$

(ii) $M_1 = X_1 \cup X_2 \cup X_4$

(iii) $M_2 = X_1 \cup X_2 \cup X_3$

(iv) $M_3 = X_3 \cup X_1 \cup X_4$

Proof. It is easy to see that $X_i \cap X_4 = \emptyset$ with $1 \leq i \leq 3$. We know that $X_1 \subseteq M_1$ since $X_1 = M_1 \setminus M_2$, and therefore $X_1 \cap (M_3 \setminus M_1) = \emptyset$. 

Figure 3.3: Vertex distributions on midsets

From lemma 3.8, we have $X_1 \subseteq M_3$, $X_2 \subseteq M_1$, and $X_3 \subseteq M_2$. So, $X_1 \cap X_2 = \emptyset$, $X_2 \cap X_3 = \emptyset$, and $X_3 \cap X_1 = \emptyset$. Moreover, since each element of $M_2$ belongs to at least one of the two sets $M_3$ and $M_1$, we know that $X_2 = M_1 \setminus M_3$ and vice-versa. Therefore,

$$M_1 = (M_1 \setminus M_3) \cup (M_1 \setminus M_2) \cup (M_1 \cap (M_2 \cap M_3)) = X_2 \cup X_1 \cup X_4$$

Similarly, we can prove the same results for $M_2$ and $M_3$.

**Corollary 3.11.** Let $\langle T, X \rangle$ be a branch decomposition of width $\beta$. Let $f_1$, $f_2$, and $f_3$ be three edges incident to the same vertex $t$ in $T$. Let $M_1 = \text{midset}(f_1)$, $M_2 = \text{midset}(f_2)$, and $M_3 = \text{midset}(f_3)$. Then the size of $M_1 \cup M_2 \cup M_3$ is at most $\frac{3}{2}\beta$.

**Proof.** Define $X_1$, $X_2$, $X_3$, and $X_4$ as in lemma 3.10. We have $M = M_1 \cup M_2 \cup M_3 = X_1 \cup X_2 \cup X_3 \cup X_4$. Since $X_1$, $X_2$, $X_3$, and $X_4$ are disjoint, $|M| = |X_1| + |X_2| + |X_3| + |X_4| = \frac{1}{2}(2|X_1| + 2|X_2| + 2|X_3| + 2|X_4|)$. Hence,
\[2|M| = 2|X_1| + 2|X_2| + 2|X_3| + 2|X_4|\]
\[\leq 2|X_1| + 2|X_2| + 2|X_3| + 3|X_4|\]
\[\leq (|X_1| + |X_2| + |X_4|) + (|X_2| + |X_3| + |X_4|) + (|X_3| + |X_1| + |X_4|)\]
\[\leq |M_1| + |M_2| + |M_3|\]
\[\leq 3\beta\]

Therefore, \(|M| \leq \frac{3}{2}\beta\).

For ease of use, we define a rooted branch decomposition as follows.

**Definition 3.12.** Given a graph \(G = (V, E)\). A rooted branch decomposition of \(G\) is a pair \((T, X)\) where \(T\) is a tree with each internal node having degree exactly three and \(X\) is a bijective mapping from the leaves of \(T\) to \(E \cup \{f_G\}\).

We can easily convert any branch decomposition to a rooted branch decomposition of the same width in constant time. We do that by creating a new leaf node \(r\) and a new internal node \(r'\). Pick any edge \((u, v)\) from the original branch decomposition, remove the edge \((u, v)\), and add three new edges \((u, r')\), \((v, r')\), \((r, r')\). The new tree is a rooted branch decomposition with the same width as the original one.

An example of converting a branch decomposition to a rooted branch decomposition is given in figure 3.4. The tree on the right is a rooted branch decomposition
Figure 3.4: Converting a branch decomposition to a rooted branch decomposition

of the branch decomposition given on the left. Nodes and edges in blue with thicker
lines are added as part of the conversion process described above.

Given a rooted branch decomposition $\langle T, X \rangle$, let $r$ be the leaf where $X(r) = f_G$.
If we take $r$ as the root of $T$, then we can view $T$ as a binary tree where each internal
node (except the root) has exactly two children. The definition of midset in this
rooted branch decomposition is still the same as for a normal branch decomposition.
We use the notion $p[i]$ to refer to the parent of node $i$ on the tree. Additionally, we
define the midset of a node $t$ on $T$ as follows.

**Definition 3.13.** Let $p[t]$ be the parent of node $t$ on a rooted branch decomposition
$\langle T, X \rangle$. The midset of node $t$ is the midset of edge $\langle t, p[t] \rangle$.

Since the root node does not have a parent, midset is undefined for the root of a
rooted branch decomposition.
Let $T_i$ be the sub-tree rooted at $p[i]$ on $T$. The sub-graph $G[i]$ of $G$ regarding a rooted branch decomposition $\langle T, X \rangle$ is defined as a graph $\langle V_i, E_i \rangle$ with $V_i = \bigcup_{k \in T_i} \text{midset}(k)$ and $E_i = \{\langle u, v \rangle | \{u, v\} \subseteq V_i \text{ and } \langle u, v \rangle \in E(G)\}$.

3.3.2 Solving Independent-Set with Branch Decompositions

Similar to path-width based and tree-width based solutions for INDEPENDENT-SET, we use dynamic programming on subsets for the branch-width based solution. Let $r'$ be the root of the given rooted branch decomposition and $r$ be the only child of $r'$. We also use a two-dimensional array $D[i, S]$ where $i \in I$ and $S \subseteq \text{midset}(i)$. $D[i, S]$ is the maximum size of all independent sets $I$ of $G[i]$ such that $I \cap \text{midset}(i) = S$. Notice that $G[r] = G$ and $\text{midset}(r) = \emptyset$. Therefore, $D[r, \emptyset]$ is the size of the maximum independent set of $G$. Hence, all we need to do is to calculate the dynamic-programming table $D$ and the answer is $D[r, \emptyset]$.

We have two difference cases when calculating $D[i, S]$.

**If $i$ is a leaf.** Let $\langle u, v \rangle = X(i)$. Therefore, $\text{midset}(i) \subseteq X(i)$. The following values (if appropriate with the actual $\text{midset}(i)$ value) will be set.

$$D[i, S] = \begin{cases} 
0 & \text{ if } S = \emptyset \\
1 & \text{ if } S = \{u\} \text{ or } S = \{v\} \\
-\infty & \text{ if } S = \{u, v\}
\end{cases}$$
If \( i \) is an internal node (non-root). Let \( j \) and \( k \) be \( i \)'s children. Let \( X_1 = \text{midset}(i) \setminus \text{midset}(j) \), \( X_2 = \text{midset}(i) \setminus \text{midset}(k) \), \( X_3 = \text{midset}(i) \cap \text{midset}(j) \cap \text{midset}(k) \), and \( Z = \text{midset}(j) \setminus \text{midset}(i) \). For each \( S \subseteq \text{midset}(i) \), let \( S_1 = S \cap X_1 \), \( S_2 = S \cap X_2 \), and \( S_3 = S \cap X_3 \). From lemma 3.10, we know that \( X_1, X_2, X_3 \), and \( Z \) are disjoint and \( \text{midset}(i) = X_1 \cup X_2 \cup X_3 \). Therefore, \( S_1, S_2, S_3 \) are disjoint and \( S = S_1 \cup S_2 \cup S_3 \). Hence, \( D[i, S] \) can be calculated by the following formula.

\[
D[i, S] = \max_{Z' \subseteq Z} \{ D[j, S_2 \cup S_3 \cup Z'] + D[k, S_1 \cup S_3 \cup Z'] - |S_3| - |Z'| \}
\]

We show that this formula is always correct.

Let \( I \) be an independent set of \( G[i] \) with maximum size such that \( I \cap \text{midset}(i) = S \). We have \( |I| = D[i, S] \) by definition. Let \( Z_0 = I \cap Z \). It is easy to see that \( I \cap X_i = S \cap X_i = S_i \) with \( 1 \leq i \leq 3 \).

Moreover, \( S_1 \cup S_3 \cup Z_0 \subseteq \text{midset}(k) \) and \( S_2 \cup S_3 \cup Z_0 \subseteq \text{midset}(j) \). Let \( I_j = I \cap V(G[j]) \) and \( I_k = I \cap V(G[k]) \). Using the same logic as in the proof for the dynamic programming formula with tree decompositions, \( I_j \) and \( I_k \) must be independent sets of \( G[j] \) and \( G[k] \) corresponding to \( D[j, S_2 \cup S_3 \cup Z_0] \) and \( D[k, S_1 \cup S_3 \cup Z_0] \), respectively. It is obvious that \( I_j \cap I_k = S_3 \cup Z_0 \). On the other hand, \( I = I_j \cup I_k \). Therefore, \( |I| = |I_j| + |I_k| - |S_3 \cup Z_0| = |I_j| + |I_k| - |S_3| - |Z_0| \).

Hence, \( D[i, S] = D[j, S_2 \cup S_3 \cup Z_0] + D[k, S_1 \cup S_3 \cup Z_0] - |S_3| - |Z_0| \).
However, the value of \( Z_0 \) is undetermined given \( i \) and \( S \). We need to try all possible values of \( Z_0 \) and pick the largest value among them. This observation gives us the formula
\[
D[i, S] = \max_{Z_0 \subseteq Z} (D[j, S_2 \cup S_3 \cup Z_0] + D[k, S_1 \cup S_3 \cup Z_0] - |S_3| - |Z_0|).
\]

Notice that \( S_1 \cup S_2 \cup S_3 = S \) and \( X_1, X_2, X_3, Z \) are disjoint. To prevent the unnecessary overhead of using set operations in the actual implementations, we can replace the formula above with the following equivalent formula \( \forall S_1 \subseteq X_1, S_2 \subseteq X_2, S_3 \subseteq X_3 : \)
\[
D[i, S_1 \cup S_2 \cup S_3] = \max_{Z' \subseteq Z} \{D[j, S_2 \cup S_3 \cup Z'] + D[k, S_1 \cup S_3 \cup Z'] - |S_3| - |Z'|\}
\]

Putting the cases together, we have algorithm 3.4 to calculate each row \( i \) of the dynamic programming table \( D \).

Finally, algorithm 3.5 uses algorithm 3.4 as a sub-routine to compute the cardinality of maximum independent sets for a graph \( G \) given a rooted branch decomposition of \( G \).

### 3.3.3 Running Time Analysis

Looking at the pseudo-code of algorithms 3.4 and 3.5, we can clearly see that the most expensive step is the for loop with all subset of \( X_1, X_2, X_3, \) and \( Z \). It runs in time \( O(2^{|X_1|+|X_2|+|X_3|+|Z|}) \). However, by lemma 3.10, \( |X_1| + |X_2| + |X_3| + |Z| = |\text{midset}(i) \cup \text{midset}(j) \cup \text{midset}(k)| \leq \frac{3}{2}k \). Hence, that step runs in time \( O(2^{\frac{3}{2}k}) \).
Algorithm 3.4 Calculating values for $D[i]$ and its descendents in a given rooted branch decomposition

**Require:** A graph $G$, a rooted branch decomposition $X$, an integer $i$.
**Ensure:** All elements $D[i]$ and its descendents in $X$ are calculated.

1: initialize $D[i]$ to $-\infty$
2: if $i$ is a leaf then
3:     $\langle u, v \rangle \leftarrow X[i]$
4:     $D[i, \emptyset] \leftarrow 0$
5:     if $u \in \text{midset}(i)$ then
6:         $D[i, \{u\}] \leftarrow 1$
7:     end if
8:     if $v \in \text{midset}(i)$ then
9:         $D[i, \{v\}] \leftarrow 1$
10: end if
11: else
12:     let $j$ and $k$ be $i$’s children
13:     run algorithm 3.4 recursively on $(G, X, j)$ and $(G, X, k)$
14:     $X_1 \leftarrow \text{midset}(i) \setminus \text{midset}(j)$
15:     $X_2 \leftarrow \text{midset}(i) \setminus \text{midset}(k)$
16:     $X_3 \leftarrow \text{midset}(i) \cap \text{midset}(j) \cap \text{midset}(k)$
17:     $Z \leftarrow \text{midset}(j) \setminus \text{midset}(i)$
18:     for all $S_1 \subseteq X_1, S_2 \subseteq X_2, S_3 \subseteq X_3, Z' \subseteq Z$ do
19:         $S \leftarrow S_1 \cup S_2 \cup S_3$
20:         $S_j \leftarrow S_2 \cup S_3 \cup Z'$
21:         $S_k \leftarrow S_1 \cup S_3 \cup Z'$
22:         $D[i, S] \leftarrow \max(D[i, S], D[j, S_j] + D[k, S_k] - |Z'|)$
23:     end for
24: end if

Algorithm 3.5 Finding maximum independent set with a rooted branch decomposition

**Require:** A graph $G$ and a rooted branch tree decomposition $X$ rooted at $r'$
**Ensure:** The cardinality of maximum independent sets of $G$.
1: $r \leftarrow r'$’s only child
2: run algorithm 3.4 on $(G, X, r)$
3: return $D[r, \emptyset]$

Therefore, algorithm 3.5 runs in time $O(h \cdot 2^{\frac{2^k}{3}})$ where $k$ is the width of the given branch decomposition and $h$ is the number of nodes in the rooted branch decomposition. Since $h = |E|$, the total running time is $O(m \cdot 2^{\frac{2^k}{3}})$. 
3.4 Tree Decompositions and Branch Decompositions

Robertson and Seymour gave a constructive proof about the following relationship between tree-width and branch-width of a graph $G$ in the tenth paper of their Graph Minor series [27].

**Lemma 3.14 ([27]).** Let $\omega(G)$ and $\beta(G)$ be the tree-width and branch-width of a graph $G$, respectively. We have $\beta(G) \leq \omega(G) + 1 \leq \frac{3}{2} \beta(G)$.

However, their definition of tree decomposition is slightly different from the definition used in this thesis. Although the two definitions are equivalent, we give a modified proof for lemma 3.14 based on the original proof in [27].

**Proof.** To show that $\omega(G) + 1 \leq \frac{3}{2} \beta(G)$, we construct a tree decomposition from an optimal branch decomposition. Let $\langle T_\beta, X_\beta \rangle$ be an optimal branch decomposition of $G$, i.e., the width of $\langle T_\beta, X_\beta \rangle$ is $\beta(G)$. We will construct a tree decomposition $\langle T_\omega, X_\omega \rangle$ where $T_\omega = T_\beta$, and $X_\omega$ is a mapping from nodes of $T_\omega$ to the power set of $V(G)$ defined as follows.

(i) If $t$ is a leaf. $X_\omega(t) = \{u, v\}$ where $\langle u, v \rangle = X_\beta(t)$.

(ii) If $t$ is an internal node. Let $f_1$, $f_2$, and $f_3$ be the three edges incident to $t$. $X_\omega(t) = M_1 \cup M_2 \cup M_3$ where $M_1 = \text{midset}(f_1)$, $M_2 = \text{midset}(f_2)$, and $M_3 = \text{midset}(f_3)$ from the branch decomposition $\langle T_\beta, X_\beta \rangle$. 
Figure 3.5: 3 nodes separate a branch decomposition tree into 7 disjoint trees

It is easy to see that $\langle T_\omega, X_\omega \rangle$ satisfies the first two conditions of tree decomposition. We use contradiction to prove that $\langle T_\omega, X_\omega \rangle$ satisfies the third condition. Assume that a vertex $v \in V$ violates the third condition, i.e., there exist three nodes $i, j, k$ where $k$ is along the path from $i$ to $j$ such that $v \notin X_\omega(k), v \in X_\omega(i)$, and $v \in X_\omega(j)$. These three nodes $i, j, k$ in $T_\beta$ separate the tree into seven disjoint trees namely $T_1, T_2, T_3, T_4, T_5, T_6$, and $T_7$ as in figure 3.5. Let $V_i$ be the vertex subset of $V$ that is the union of all leaves in $T_i$ with $1 \leq i \leq 7$. Notice that $T_6$ and $T_7$ are formed by the edges and nodes on the path from $i$ to $k$ and the path from $k$ to $j$, respectively. Therefore, $T_6$ and $T_7$ can be empty trees.

Since $v \in X_\omega(i), v$ belongs to at least one of the two sets $V_1$ and $V_2$ according to corollary 3.9. Similarly, since $v \in X_\omega(j), v$ belongs to at least one of the two sets $V_3$ and $V_4$. Hence, $v$ belongs to $V_1 \cup V_2$ and $V_3 \cup V_4$. Let $f$ be the lower left edge incident to $k$ in figure 3.5. By definition 3.6, $\text{midset}(f) = (V_1 \cup V_2 \cup V_6) \cap (V_3 \cup V_4 \cup V_5 \cup V_7) \supset
(V_1 \cup V_2) \cap (V_3 \cup V_4) \ni v. \text{ Hence, } v \in \text{midset}(f) \subseteq X_\omega(k). \text{ This contradicts the assumption we made before. Hence, } \langle T_\omega, X_\omega \rangle \text{ satisfies the third condition of a tree decomposition.}

Therefore, \langle T_\omega, X_\omega \rangle \text{ is a tree decomposition of } G. \text{ As a result of the construction of the bags in } T_\omega, \text{ the size of each bag in } T_\omega \text{ is at most } \frac{3}{2}\beta(G) \text{ by corollary 3.11. This means } \omega(G) \leq \frac{3}{2}\beta(G) - 1.

To show that } \beta(G) \leq \omega(G) + 1, \text{ a branch decomposition is built from any given optimal tree decomposition. Let } \langle T_\omega, X_\omega \rangle \text{ be an optimal tree decomposition of } G, \text{ i.e. the width of } \langle T_\omega, X_\omega \rangle \text{ is } \omega(G). \text{ W.L.O.G., we can assume that } G \text{ does not have any isolated vertices and the leaf bags in } T_\omega \text{ are of size at least two. We will transform } \langle T_\omega, X_\omega \rangle \text{ to a new tree decomposition where each internal node has exactly two children, each leaf bag is of size exactly two, and there is a bijective mapping from the leaf bags to } E(G).

\textbf{Step 1.} \text{ If there is an edge } e = \langle u, v \rangle \in E(G) \text{ without a corresponding leaf bag in } T_\omega, \text{ let } T_i \text{ be the bag in } T_\omega \text{ where } \{u, v\} \subseteq T_i. \text{ We create a leaf node } T_k = \{u, v\} \text{ in } T_\omega \text{ and make } T_i \text{ the parent of } T_k.

\text{By repeating this step, we can guarantee that for each edge } e = \langle u, v \rangle \in E(G), \text{ there is at least one leaf bag } T_i \text{ in } T_\omega \text{ such that } T_i = \{u, v\}. \text{ And obviously, the modified tree after this step is still a tree decomposition of } G.
Step 2. If there are two leaf bags $T_i$ and $T_j$ in $T_\omega$ corresponding to the same edge $e = \langle u, v \rangle \in E(G)$, remove one of them. Remove all leaf bags that do not have a corresponding edge in $E(G)$.

By repeating this step, we guarantee that for each edge $e = \langle u, v \rangle \in E(G)$, there is exactly one leaf bag $T_i$ in $T_\omega$ such that $T_i = \{u, v\}$. Additionally, there are no other leaf bags in $T_\omega$.

This is true because if there is an edge $e = \langle u, v \rangle \in E(G)$ such that there exists two leaf bags $T_i$ and $T_j$ in $T_\omega$ where $T_i = T_j = \{u, v\}$. According to the third condition of a tree decomposition, the parent of $T_j$ also contains $u$ and $v$. Therefore, we can remove bag $T_j$ without violating any of the three conditions for a tree decomposition. Hence, there is exactly one leaf bag in $T_\omega$ for each edge in $E(G)$.

On the other hand, if there is a leaf bag $T_i$ in $T_\omega$ of size two that does not correspond to an edge in $E(G)$, let $\{u, w\} = T_i$. This means $u$ and $w$ are not connected in $G$. Since there is no isolated vertices in $G$, $u$ and $w$ must be in some other leaves, hence the parent of $T_i$ must contain both $u$ and $w$. Therefore, we can remove bag $T_i$ without violating any of the tree decomposition conditions.

Similarly, if there is a leaf bag $T_i$ in $T_\omega$ of size $l \geq 3$, let $\{u_1, u_2, ..., u_l\} = T_i$. Since $G$ does not have isolated vertices, each of the vertices $u_j$ must be in some other leaf bag. By the third condition of a tree decomposition, the parent of $T_i$ must contain $u_j$. Hence, the parent of $T_i$ is a superset of $T_i$. Therefore, we can remove bag $T_i$ from the tree and still have a tree decomposition.
**Step 3.** If there is an internal node $T_i$ with more than two children, let $T_{i_1}, T_{i_2}, \ldots, T_{i_l}$ be $T_i$’s children. Create a new node $T_i'$ and make $T_i$ the parent of $T_i'$. Make the first $(l - 1)$ children of $T_i$ children of the newly created node $T_i'$.

By repeating this step, we guarantee that each internal node in $T_\omega$ has at most two children.

**Step 4.** If there is an internal node $T_i$ with exactly one child $T_k$ and $T_i$’s parent is $T_j$. Remove $T_i$ and make $T_j$ the parent of $T_k$.

By repeating this step, we guarantee that each internal node in $T_\omega$ has exactly two children while $T_\omega$ still satisfies three conditions of a tree decomposition.

We prove that the claim above is true. From step 3, each internal node of $T_\omega$ has at most two children, so we only need to modify the ones with one child. Let $T_i$ be such node. If $T_i$ is the root, we can simply remove $T_i$ and make its child the new root. Otherwise, let $T_j$ be the parent of $T_i$, and $T_k$ be the child of $T_i$. We will prove that removing bag $T_i$ from $T_\omega$ and make $T_j$ the parent of $T_k$ will still give us a tree decomposition of $G$. Since each edge in $E(G)$ has a corresponding leaf bag in $T_\omega$, the first and second conditions of tree decomposition always hold when we remove an internal node. The third condition also holds because any path containing $T_i$ will also go through $T_k$ and $T_j$.

**Step 5.** Finally, remove the root of $T_\omega$ and create an edge between its two children to make all internal nodes of $T_\omega$ degree three.
Since the root has exactly two children, we can use the same logic when removing
a non-root internal node with exactly one child from step 4.

At this point, each non-leaf vertex in $T_\omega$ has degree exactly three, and there is
a bijective mapping $X_\beta$ from the leaves of $T_\omega$ to $E(G)$. Hence, $\langle T_\omega, X_\beta \rangle$ is a branch
decomposition.

Let $f$ be an edge connecting $T_i$ and $T_j$ in $T_\omega$. It is easy to see that $\text{midset}(f) \subseteq X_i$
and $\text{midset}(f) \subseteq X_j$. Therefore, $|\text{midset}(f)| \leq \max(|X_i|, |X_j|) \leq \omega(G) + 1$. Hence,
the width of the constructed branch decomposition is at most $\omega(G) + 1$. That implies
$\beta(G) \leq \omega(G) + 1$.

Notice that the proof presented above is constructive. Therefore, given any branch
decomposition of width $B$, we can build a tree decomposition of width at most $\frac{3}{2}B - 1$
in polynomial time. Similarly, given any tree decomposition of width $T$, we can build
a branch decomposition of width at most $T + 1$ in polynomial time.
Chapter 4

Optimal Branch Decompositions for Planar Graphs

Although the problem of computing branch-width for general graphs is NP-Complete, we can solve it efficiently in polynomial time if the input graph is restricted to a planar graph. In 1994, Seymour and Thomas [29] gave an $O(n^2)$ algorithm for the decision version of the branch-width problem for planar graphs:

**Branch-Width**

| Input: | A planar graph $G = (V, E)$ with $|V| = n$ and an integer $k$. |
|--------|---------------------------------------------------------------|
| Output: | Does $G$ have branch-width $\leq k$? |

This algorithm uses another type of graph decomposition called a carving decomposition. It checks whether $G$ has branch-width $k$ by verifying if its medial graph has carving-width $2k$. This approach is based on an important theorem in their paper [29]: the branch-width of a connected planar graph $G$ with at least two edges is exactly half the carving-width of its medial graph.

Seymour and Thomas also gave an algorithm to construct an optimal branch decomposition for planar graphs. To construct an optimal branch decomposition, the algorithm for the decision problem is called $O(m \cdot n)$ times where $m$ is the number of edges and $n$ is the number of vertices in the graph. The total running time of the constructive algorithm is $O(n^4)$ due to the fact that $m = O(n)$ for planar graphs.
4.1 Preliminaries

4.1.1 Planar Graphs

Planar graph is a well-known concept used widely in graph theory. A graph is planar if we can find a way to draw its vertices and edges on a plane without crossing the edges except at their endpoints. A formal definition of planar graphs is as follows.

**Definition 4.1.** A graph $G$ is planar if it can be drawn on a plane where vertices correspond to points and edges correspond to curves or lines connecting its endpoints such that edges only intersect at the endpoints.

Notice that the embeddings of a graph are not unique.

![Different embeddings for the same graph](image)

**Figure 4.1:** Different embeddings for the same graph

Dual graph is a notion related to planar graph. There are a variety of equivalent definitions for the dual graph of a planar graph. Here we give a simple definition of dual graph that can be used easily throughout this thesis.
Definition 4.2 (e.g. [33]). Let $G$ be a planar graph with a given embedding $H$. The dual graph of $G = \langle V, E \rangle$ is a graph $G_d = \langle V_d, E_d \rangle$ where there is a bijective mapping $X_v$ from $V_d$ to the set of faces in $H$ and a bijective mapping $X_e$ from $E_d$ to $E$ such that $e_d = \langle u_d, v_d \rangle \in E_d$ if and only if the faces $X_v(u_d)$ and $X_v(v_d)$ share an edge $X_e(e_d)$ in $H$.

With a fixed embedding, the dual graph for a given graph $G$ is unique by definition 4.2. Since there can be different embeddings for the same graph, dual graphs are not unique.

Figure 4.2: The same graph has two dual graphs that are not isomorphic

Notice that the dual graph is also planar even though it may be a multigraph (a graph that has self-loops or multiple edges between the same vertices).

From now on, we will use $G^*$ to denote the dual graph of $G$, $e^*$ to denote the corresponding edge in $E(G^*)$ of an edge $e \in E(G)$, $r^*$ to denote corresponding vertex
in $V(G^*)$ of a face $r$ in $G$, and $v^*$ to denote the corresponding face in $G^*$ of a vertex $v$ in $G$.

Given an embedding of a planar graph, an angle is defined as follows.

**Definition 4.3 ([29]).** Let $G$ be a planar graph with a given embedding $H$. For each vertex $v$ in $G$, let $d$ be its degree and let $e_1, e_2, \ldots, e_d$ be the edges incident on $v$ in counter clockwise order in $H$. For every $1 \leq i \leq d$, the pair $\langle e_i, e_{i+1} \rangle$ forms an angle in the embedding $H$ with $e_{d+1} = e_1$. When $d = 1$, there is exactly one angle between edge $e_1$ to itself.

It is obvious that the number of angles in an embedding is equal to the sum of the degree of all vertices.

**Definition 4.4 ([29]).** Let $G$ be a planar graph with a given embedding $H$. The medial graph of $G = \langle V, E \rangle$ is a graph $G_m = \langle V_m, E_m \rangle$ where there is a bijective mapping $X_e$ from $V_m$ to $E$ and there is an edge in $E_m$ connecting $u_m$ and $v_m \in V_M$ if there is an angle in $H$ between edges $X_e(v_m)$ and $X_e(u_m)$.

Figure 4.3 demonstrates how to construct the medial graph $G_m$ from a planar graph $G$. It is trivial to see that there is also a bijective mapping from $E_m$ to the set of angles in $H$. Since the number of angles in $H$ equals to the sum of degree for all vertices in $G$ and the sum of degree for all vertices in $G$ is exactly $2 \cdot |E|$, we have $|E_m| = 2 \cdot |E| = 2 \cdot |V_m|$.
Similar to dual graphs, the medial graphs of $G$ are not unique because they depend on the given embedding. Medial graphs are also planar. Moreover, there is a bijective mapping from the set of faces in a medial graph $G_m$ to the set of vertices and faces in the original graph $G$. 

Figure 4.3: An example of how to construct medial graph
4.1.2 Carving and Carving-width

**Definition 4.5 ([29]).** Given a graph $G = \langle V, E \rangle$. A carving decomposition of $G$ is a pair $\langle T, X \rangle$ where $T$ is a tree with each internal node having degree exactly three and $X$ is a bijective mapping from the leaves of $T$ to $V$.

We can see that a carving decomposition is very similar to a branch decomposition except its leaves are mapped to the set of vertices instead of the set of edges. It follows that the number of non-leaf nodes in a carving decomposition is $|V| - 2$.

Given a carving decomposition $\langle T, X \rangle$, the midset of each edge $f$ in $T$ is defined as follows.

**Definition 4.6.** Let $T_1$ and $T_2$ be the two components of $T \setminus \{f\}$. The midset of $f$ is the set of all edges $e = \langle u, v \rangle$ in $E$ such that there exist two leaves $l_1 \in T_1$ and $l_2 \in T_2$ and $X(l_1) = u$ and $X(l_2) = v$.

From the notion of midsets, the width of a carving decomposition is defined by taking the maximum size of any midsets in it.

**Definition 4.7.** Let $\langle T, X \rangle$ be a carving decomposition of $G$. The width of $\langle T, X \rangle$ is $\max_{f \in T} \{|\text{midset}(f)|\}$.

The carving-width of a graph $G$ is the minimum width of all carving decompositions of $G$. 
4.1.3 Branch Decompositions and Carving Decompositions

Let $G$ be a planar graph, $G_m$ be its medial graph, and $\langle T, X \rangle$ be a carving decomposition of $G_m$. From definition 4.4, there exists a bijective mapping $X_e$ from $V(G_m)$ to $E(G)$. Since both $X_e$ and $X$ are bijective and the range of $X$ is the same as the domain of $X_e$, $X_e \circ X$ is also a bijective mapping from the leaves of $T$ to $E(G)$. Therefore, $\langle T, X_e \circ X \rangle$ is a branch decomposition of $G$.

Moreover, the width of $\langle T, X_e \circ X \rangle$ is at most the width of $\langle T, X \rangle$. We can prove this by comparing the number of vertices in the original graph $G$ shared between two edges $e_1$ and $e_2 \in E(G)$ with the number of edges in $G_m$ connecting $X^{-1}(e_1)$ and $X^{-1}(e_2)$.

If edges $e_1$ and $e_2$ do not share an endpoint, the number of vertices shared between $e_1$ and $e_2$ is zero. In this case, the number of edges in $G_m$ connecting $X^{-1}(e_1)$ and $X^{-1}(e_2)$ is also zero.

![Diagram of original graph and medial graph](image)

Figure 4.4: Two edges incident to a vertex of degree 2 in the original graph
If \( e_1 \) and \( e_2 \) share only one endpoint \( u \), we have two cases. When \( u \) is of degree exactly 2, there are exactly two edges in \( G_m \) connecting \( X^{-1}(e_1) \) and \( X^{-1}(e_2) \) (figure 4.4). When \( u \) is of degree \( > 2 \), there is exactly one edge connecting \( X^{-1}(e_1) \) and \( X^{-1}(e_2) \) in \( G_m \) (figure 4.5).

![Figure 4.5: Two edges incident to a vertex of degree > 2 in the original graph](image)

In the last case, when \( e_1 \) and \( e_2 \) share both of its endpoints \( u \) and \( v \), i.e., \( e_1 \) and \( e_2 \) are multiple edges connecting the same pair of vertices. In this case, there are two, three, or four edges in \( G_m \) connecting \( X^{-1}(e_1) \) and \( X^{-1}(e_2) \), where the actual number depends on the degrees of \( u \) and \( v \) in \( G \) (figure 4.6).

In every case, there is at least one edge in the medial graph \( G_m \) for each vertex in the original graph \( G \) shared between two edges. Therefore, the width of the branch decomposition \( \langle T, X_e \circ X \rangle \) for \( G \) is at most the width of the original carving decomposition \( \langle T, X \rangle \) for \( G_m \).

Note, however, that the relationship between carving decompositions and branch decompositions given above is not optimal. When the given carving decomposition
Three different cases for multiple edges \( e_1 \) and \( e_2 \) in graph \( G \) between two vertices \( u \) and \( v \). In each figure, graph \( G \) is on the left and its medial graph \( G_m \) is on the right. The edges we are counting are drawn with thicker lines in \( G_m \).

Figure 4.6: Multiple edges in the original graph

is an optimal one, its width is exactly twice of the width of the constructed branch decomposition. In other words, there is a close relationship between the branch-width of a graph and the carving-width of its medial graph. Seymour and Thomas [29] gave a complete proof for the following result in their paper in 1994.

**Lemma 4.8 ([29]).** Let \( G \) be a connected planar graph with at least two edges and let \( M \) be its medial graph. The branch-width of \( G \) is exactly half the carving width of \( M \).
Using this result, the algorithm deciding whether a given planar graph $G$ has branch-width at least $k$ actually checks if the medial graph $G_m$ of $G$ has carving-width at least $2k$. Moreover, we can easily construct an optimal branch decomposition for $G$ if an optimal carving decomposition is found for $G_m$ as described above. Therefore, the algorithms for computing branch-width and building optimal branch decompositions for planar graphs given in [29] essentially deal with carving-width and optimal carving decompositions for medial graphs of the input.

4.2 The Carving-width Decision Problem and the Rat-catching Game

In this section, we assume that the input graph $G$ for the CARVING-WIDTH problem is a medial graph.

<table>
<thead>
<tr>
<th>CARVING-WIDTH</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A planar graph $G = \langle V, E \rangle$ with $</td>
</tr>
<tr>
<td><strong>Output:</strong> Does $G$ have carving-width $\geq k$.</td>
</tr>
</tbody>
</table>

The algorithm given by Seymour and Thomas, which is commonly referred to the ST-PROCEDURE, solves the decision problem CARVING-WIDTH above in $O(n^2)$ steps. The carving-width problem is related to a game called rat-catching [29]. Seymour and Thomas gave the following relationships between the rat-catching game and carving-width.
Lemma 4.9 ([29]). *The rat-catcher can win the “game” if and only if $G$ has degree $< k$ and its carving-width is $< k$.*\(^1\)

Therefore, we can decide whether the carving-width of a given planar graph is at least $k$ by solving the rat-catching game.

### 4.2.1 The Rat-catching Game

The rat-catching game was introduced in the same paper by Seymour and Thomas [29] that described how to compute the branch-width for planar graphs in polynomial time. There are two players in the rat-catching game: the rat-catcher and the rat. The game happens in a house with plenty of rooms and rooms share walls with others. The house can be viewed as a planar graph where edges are walls, vertices are room corners and faces are rooms. The rat-catcher always makes noises at a given level $k$.

At the beginning of the game, the rat-catcher stays in a room of his choice and the rat stays in some corner of its choice. The rat-catcher and the rat take turns in the game and the rat-catcher always moves first. The game goes on unless the rat-catcher wins. The legal moves in the game are described as follows:

**Step 1** If the rat-catcher is in a room (face), he moves to one of the walls of the room (edges of the face). If he is on a wall (edge) that is incident to rooms (faces) $A$ and $B$, he moves to room (face) $B$ assuming that he entered this wall (edge) from room (face) $A$.

\(^1\)Note: If $G$ has a vertex of degree $\geq k$, its carving width is $\geq k$. 
(Step 2) If the rat is currently in corner (vertex) $u$, it moves to corner (vertex) $v$ where there is a path from $u$ to $v$ in $G$ and every edge in that path is “quiet”.

(Step 3) The rat-catcher wins if the rat is in a corner (vertex) $u$ with less than $k$ incident walls (edges) and the rat-catcher is in a room (face) incident with $u$. Otherwise, repeat from step 1.\(^2\)

\(^{2}\)If there is a vertex $v$ of degree $\geq k$, the rat can remain in $v$ forever; the rat-catcher will never win.
An edge $e$ in $G$ is quiet if there is no closed path of length $< k$ in $G^*$ that contains both the edge $e^*$ and the vertex $r^*$ when the rat-catcher is in room $r$, or both edges $e^*$ and $f^*$ when the rat-catcher is on wall $f$.

Notice that by the aforementioned rules, the rat-catcher actually moves between vertices and edges of the dual graph. As demonstrated in figure 4.7, the rat moves between black vertices and black edges, while the rat-catcher moves between blue vertices and blue dotted edges. Figure 4.8 shows an example of the possible moves for the rat when the noise level is 4 and the rat-catcher is on the right most edge of the graph from figure 4.7 with all noisy edges marked as red.
the black graph. The dual graph is used to determine which edges are noisy. These noisy edges in the dual graph are marked as red. The corresponding edges in the original graph (black graph) are also marked red accordingly.

Locations of the rat-catcher are colored brown and locations of the rat are colored green.

Figure 4.9: An example of the rat-catching game
If we remove all the noisy (red) edges from the original graph of figure 4.8, the remaining graph is separated into three connected components. In this case, the rat can only move to vertices in the same component that it is currently in.

Figure 4.9 shows an example of a rat-catching game with noise level $k = 5$. The first figure describes the original graph in black with its dual in dotted blue lines. Locations of the rat-catcher are colored brown while locations of the rat are colored green. In each step, noisy edges are colored red. In this simple example, all the edges are noisy because they are in closed paths of length at most 4. Therefore, the rat-catcher simple moves across faces and edges towards the rat and finally wins the game after a few steps.

Note that if the noise level $k$ is 4, the rat-catcher will never win the game since there is always one quiet edge when he is on one of the internal edges (edges incident to $D$).

### 4.2.2 The Rat-catching Algorithm

**Rat-Catching**

**Input:** A planar graph $G = \langle V, E \rangle$ with $|V| = n$ and an integer $k$.

**Output:** Can the rat-catcher win the rat-catching game on graph $G$ with noise level $k$?

In order to solve the rat-catching problem, we first need to build the game state graph with vertices indicating the positions of the rat-catcher and the rat. We can denote $RC(r, v)$ as the state where the rat-catcher stays in face $r$ and the rat is in corner $v$, and $RC(e, v)$ as the state where the rat-catcher stays in wall $e$ and the rat
is in corner \( v \). There is an edge between \( RC(r, v) \) and \( RC(e, v) \) if \( e \) is incident with \( r \) (this indicates a rat-catcher move). Similarly, there are an edge between \( RC(e, v_1) \) and \( RC(e, v_2) \) and an edge between \( RC(r, v_1) \) and \( RC(r, v_2) \) if there is a quiet path between \( v_1 \) and \( v_2 \) in \( G \) with regards to \( e \) and \( r \), respectively.

Notice that for a given edge \( e \), if we have three vertices \( v_1, v_2, \) and \( v_3 \) such that \( RC(e, v_1) \) is adjacent to \( RC(e, v_2) \) and \( RC(e, v_2) \) is adjacent to \( RC(e, v_3) \) then \( RC(e, v_1) \) and \( RC(e, v_3) \) are adjacent. Furthermore, let \( G_e \) be the graph \( G \) after removing all the noisy edges (assuming the rat-catcher is in edge \( e \)), and \( \{C_i\}_{i=1}^t \) be the set of connected components of \( G_e \) where \( t \) is the number of components in \( G_e \).

Then, for each \( 1 \leq i \leq t \), \( RC(e, v_j) \) with all \( v_j \in C_i \) form a clique in the game state graph. Hence, we can group all states \( RC(e, v_j) \) into one big state \( RC(e, i) \) (with \( i \) indicating the component index of \( G_e \)) to make the game state graph a bipartite graph. Moreover, each state now represents the game state when the rat-catcher makes the next move.

In order to solve the rat-catcher problem, we will start by marking \( RC(r, v) \) as a losing state (for the rat) if \( v \) is incident with \( r \) according to the game rules. From these losing states, we will do a graph traversal and mark other states as losing states.

The strategy of marking losing states is described in detail in [4]. \( RC(e, i) \) is a losing state if \( r \) is a face incident to \( e \) and \( RC(r, v) \) is a losing state for all \( v \in C_i \) of \( G_e \). This is true because the rat-catcher can follow the following strategy to win the game. Let \( r \) and \( r' \) be the two faces incident to \( e \). If the rat-catcher can move
from $e$ to $r$ (because he moved to $e$ from $r'$) then obviously he wins the game. The rat-catcher can still win the game even if he has to move from $e$ to $r'$ (because he came to $e$ from $r$). At that time, the rat is still at some vertex in $C_i$ and it is the rat-catcher’s turn. The rat-catcher then can move back to $e$ preventing the rat from moving out of $C_i$. In the next turn, the rat-catcher then moves to $r$ and it comes back to the first case.

On the other hand, if $RC(e, i)$ is a losing state, then for both faces $r$ and $r'$ that are incident on $e$ and for all vertices $v \in C_i$ of $G_e$, $RC(r, v)$ and $RC(r', v)$ are losing states. This is simply because if the rat-catcher is at $r$ while the rat is at $v$, the rat-catcher can just move to edge $e$. At this time, since $v \in C_i$ of $G_e$, the rat cannot move to any vertex outside of $C_i$ because the rat catcher is on edge $e$. Therefore, the game state will be $RC(e, i)$ regardless of where the rat goes. From this point, the rat will eventually lose the game. Hence, $RC(r, v)$ is a losing state.

If the rat-catcher can win the game, every state in the game state graph will eventually be marked. Moreover, if every state for a face $r$ is marked, then the rat-catcher will win the game. This is because every game state for edge $e$, where $e$ is incident to $r$, will also be marked by the strategy above. Consequently, if we let $r'$ be the other face incident to $e$, then every game state for face $r'$ incident to $e$ will also be marked by the second rule of the strategy. This marking will spread out and it will be the case that eventually all game states for every face and edge will be marked. Similarly, if every state for an edge $e$ is marked, the rat-catcher will win the game.
This observation is very helpful in speeding-up the algorithm because we do not have to wait until all the game states are marked to return the answer YES. Instead, we can stop as soon as every state for an edge of a face are marked and return the answer YES.

4.2.3 The ST-Procedure

The ST-Procedure follows the strategies given in the previous sections to determine whether a graph has carving-width \( k \). The pseudo-code for this algorithm is given as algorithm 4.1.

Notice that there is an additional step at the beginning of ST-Procedure to compute all-pair shortest paths matrix \( D \) for \( G^* \). This matrix is used to check whether an edge \( f = \langle w, t \rangle \) is noisy from edge \( e = \langle u, v \rangle \) with the given noise level \( k \) by verifying if:

\[
D[u^*, w^*] + D[t^*, v^*] + 2 < k \quad \text{or} \quad D[u^*, t^*] + D[w^*, v^*] + 2 < k.
\]

As mentioned in the previous section, to speed up the marking process, we can check if all states for an edge or a face are marked whenever we change the value of \( RC[e, i] \) or \( RC[r, v] \) to marked. This technique may speed up the running time of ST-Procedure but it will not change the asymptotic running time of this procedure.
**Algorithm 4.1 The ST-Procedure**

**Require:** A planar graph \( G \) and an integer \( k \).

**Ensure:** Return YES if the rat-catcher can win the game on \( G \) with noise level \( k \).
Otherwise, return NO.

1: **construct** the embedding and the dual graph \( G^* \) for \( G \)
2: return NO if degree of \( G \) \( \geq k \)
3: **find** all-pair shortest paths \( D \) in \( G^* \)
4: **initialize** all the game states as un-marked
5: for each edge \( e \) of \( G \) and connected component \( C_i \) of \( G_e \) do
   6: set \( \text{CountUnmarkedNeighbour}[e,i,r] \) to \( |C_i| \) for each face \( r \) incident with \( e \)
7: end for
8: **initialize** the \( \text{ProcessLosingStack} \) as an empty stack
9: for each face \( r \) of \( G \) and vertex \( v \) incident to \( r \) do
   10: push \((r,v)\) into \( \text{ProcessLosingStack} \) and mark \( RC[r,v] \)
11: end for
12: while \( \text{ProcessLosingStack} \) is not empty do
13: pop \( \text{aState} \) from \( \text{ProcessLosingStack} \)
14: if \( \text{aState} \) is a face state \((r,v)\) then
   15: for each edge \( e \) of face \( r \) do
      16: let \( i \) be the component of \( G_e \) containing \( v \)
      17: decrease \( \text{CountUnmarkedNeighbour}[e,i,r] \) by one
      18: if \( \text{CountUnmarkedNeighbour}[e,i,r] = 0 \) then
         19: push \((e,i)\) into \( \text{ProcessLosingStack} \) and mark \( RC[e,i] \)
      end if
   end for
12: else if \( \text{aState} \) is an edge state \((e,i)\) then
   21: for each face \( r \) incident with \( e \) and vertex \( v \) in \( C_i \) do
      22: if \( RC[r,v] \) is not marked then
         23: push \((r,v)\) into \( \text{ProcessLosingStack} \) and mark \( RC[r,v] \)
      end if
   end for
24: end if
24: end while
30: if all the game states are marked then
31: return YES
32: else
33: return NO
34: end if
4.2.4 Running Time Analysis

Since $G$ is an unweighted graph, we can simply use breadth-first-search to calculate the all pair shortest path matrix $D$. The running time of breadth-first-search is $O(|V| \cdot |E|) = O(|V|^2)$ because $|E| = O(|V|)$ due to planarity.

Because there are $O(|V|^2)$ states in the game state graph, the body of the while loop will be executed at most $O(|V|^2)$ time. We will analyze the amortized cost for each iteration.

When the processing state is of type $(r, v)$, each edge $e$ of face $r$ is processed once. It follows that the amortized cost for each face state is $O(\sum_{\forall \text{face } j} D_j \cdot |V|/|V|^2)$ where $D_j$ is the number of edges belonging to face $j$. However, each edge in a planar graph belongs to two faces (one face on the left and one face on the right of the edge). Therefore, $\sum_{\forall \text{face } j} D_j = 2 \cdot |E|$. Hence, the amortized cost for processing each face state is $O(2 \cdot |E| \cdot |V|/|V|^2) = O(|E|/|V|) = O(1)$.

When the processing state is of type $(e, i)$, each vertex $v$ in $C_i$ will be processed at most twice (once for each face incident with $e$). Hence, the amortized cost for each iteration in this case is $O(\sum_{e \in E} \sum_{1 \leq i \leq t_e} |C_i|/|V|^2)$ where $t_e$ is the number of connected components for $G_e$. Since $\sum_{1 \leq i \leq t_e} |C_i| = |V|$ and $|E| = O(|V|)$, the amortized cost for processing each edge state is $O(1)$.

Overall, the total running time of ST-Procedure is $O(|V|^2)$. 
4.3 The Optimization Problem

In order to find the carving-width of a given planar graph, we can directly use ST-PROCEDURE to try all possible values from 1 to $|V|$ until it returns YES. This simple approach gives us a $O(k \cdot |V|^2)$ running time where $k$ is the actual value of the carving-width. Without any knowledge about how large the carving-width can be, the running time is $O(|V|^3)$.

To speed up the running time, we can use binary search instead of linear search on the value of carving-width. Since carving-width is at most $|E| = O(|V|)$, the running time will boost up to $O(|V|^2 \cdot \log |V|)$.

Algorithm 4.2 Computing the carving-width of a planar graph with binary search

Require: A planar graph $G$.
Ensure: Returns $k$ – the carving-width of $G$.

1: $low \leftarrow$ be the maximum degree in $G$
2: $hi \leftarrow |V|$
3: while $low < hi$ do
4: \hspace{1em} $mid \leftarrow (low + hi)/2$
5: \hspace{1em} if ST-PROCEDURE (algorithm 4.1) on $(G, mid)$ returns YES then
6: \hspace{2em} $low \leftarrow mid$
7: \hspace{1em} else
8: \hspace{2em} $hi \leftarrow mid - 1$
9: \hspace{1em} end if
10: end while
11: return $(low + hi)/2$

However, we still have not taken advantage of the knowledge about the upper bounds for branch-width and carving-width. It is proven that branch-width for planar graphs are bounded by $\alpha \sqrt{|V|}$. The best known upper bound for branch-width for planar graphs is due to Fomin and Thilikos [15] by the following theorem.
Theorem 4.10 ([15]). For any planar graph $G$, branch-width of $G$ is at most $\sqrt{4.5 \cdot |V(G)|}$.

Let $G$ be the original planar graph with branch-width $\beta$ and $G_m$ be its medial graph with which we are applying ST-PROCEDURE. Let $v, e, f, v_m, e_m,$ and $f_m$ be the number of vertices, number of edges, number of faces for $G$ and $G_m$, respectively. By Euler’s formula, we have $v = e - f + 2$. On the other hand, we know that $f_m = v + f$, $v_m = e$, and $e_m = 2 \cdot v_m$ from definition 4.4. Hence,

$$\beta \leq \sqrt{4.5 \cdot |V(G)|}$$

$$= \sqrt{4.5 \cdot v}$$

$$= \sqrt{4.5 \cdot (e - f + 2)}$$

$$\leq \sqrt{4.5 \cdot (e + 1)}$$

$$\leq \sqrt{4.5 \cdot (v_m + 1)}$$

Therefore, the carving-width for a connected planar graph $G$ is bounded by $2\sqrt{4.5 \cdot (|V(G)| + 1)}$. While this bound can be used to speed up the running time by a constant factor, it does not change the overall asymptotic running time of the algorithm.
4.4 The Constructive Algorithm

4.4.1 The Algorithm

Given a graph $G = \langle V, E \rangle$. Let $e = \langle u, v \rangle$ be an edge in $E$. The operation of removing $e$, replacing $u$ and $v$ with a new vertex $t$, and making all edges incident to $u$ or $v$ incident to $t$ is called contracting edge $e$ on $G$ [4]. Here, we denote the resulting graph after the edge contraction of $e$ on $G$ as $G \setminus e$. Notice that the new graph $G \setminus e$ may contain multiple edges connecting the same endpoints. \(^3\)

An example of an edge contraction is shown in figure 4.10. The edge connecting two endpoints A and B (in blue) is contracted forming the vertex $X$ in the graph on the left.

![Original Graph and After Contraction](image)

Figure 4.10: Graph after an edge contraction

Seymour and Thomas [29] gave the constructive algorithm by using the ST-

\begin{procedure}
\end{procedure}

based on the observation: if a 2-connected graph $G$ has carving-width

\(^3\)Notice that, in this section, we use the \setminus symbol to denote edge contractions. This should not be confused with the set theoretic operation set minus.
< k, then there exists an edge \( e \in E(G) \) such that the \( G \setminus e \) also has carving-width \(< k\). Moreover, given the carving decomposition \( \langle T', X' \rangle \) of \( G' = G \setminus e \) with \( e = \langle u, v \rangle \in E(G) \) and \( t \) being the new vertex, we can construct the carving decomposition \( \langle T, X \rangle \) of \( G \) by adding two leaves \( L_u \) and \( L_v \) corresponding to \( u \) and \( v \) to the carving decomposition of \( G \setminus e \) and then make the leaf node corresponding to \( t \) the parent of both \( L_u \) and \( L_v \). The new tree has every internal node of degree exactly three and each leaf node corresponding to a vertex in \( V \). Therefore, it is a carving decomposition of \( G \).

By this construction, we show that the carving decomposition \( \langle T, X \rangle \) for \( G \) has width \(< k \). It is easy to see that the size of the midsets for old edges are still the same. We only need to justify the size of the midsets for leaves \( L_u \) and \( L_v \) in the new carving decomposition. Let \( cw' \) be the width of \( \langle T', X' \rangle \). As a result of the construction of \( \langle T', X' \rangle \), we know \( cw' < k \). Let \( d_u \) and \( d_v \) be the degree of \( u \) and \( v \) in \( G \), respectively. Since \( L_u \) and \( L_v \) are leaves, there is only one edge incident to each of these nodes. By definition 4.6, the sizes of midsets for edges incident to \( L_u \) and \( L_v \) in the carving decomposition are \( d_u \) and \( d_v \), respectively. Moreover, we know that the carving-width of \( G \) is \(< k \), hence the degree of any vertex in \( G \) is \(< k \). That means \( d_u < k \) and \( d_v < k \). Therefore, the width of \( \langle T, X \rangle \) is \( \max(d_u, d_v, cw') < k \).

Finally, if \( G \) only has three vertices, the carving decomposition of \( G \) is unique. More specifically, the carving decomposition for \( G \) has exactly one internal node adjacent to three leaves. Each leaf node corresponds to one vertex of \( G \). This observation
serves as the base case for the constructive algorithm because as we find and contract edges, the number of vertices in the remaining graph decreases.

Putting these all together, we have algorithm 4.3 to construct an optimal carving decomposition for a given 2-connected planar graph $G$.

**Algorithm 4.3** Constructing an optimal carving decomposition of a 2-connected planar graph

**Require:** A 2-connected planar graph $G$

**Ensure:** An optimal carving decomposition of $G$

1. use algorithm 4.2 to find carving-width $k$ of $G$
2. initialize $\text{contractionStack}$ to empty
3. while $|V(G)| > 3$ do
   4. for all edge $e = \langle u, v \rangle \in E(G)$ do
      5. if $\text{ST-Procedure}(G \setminus e, k)$ returns YES then
         6. let $t$ be the new vertex in $G \setminus e$
         7. push $\langle \langle u, v \rangle, t \rangle$ to $\text{contractionStack}$
         8. $G \leftarrow G \setminus e$
         9. break out of the for loop
      10. end if
   11. end for
3. end while
13. $\text{decomposition} \leftarrow$ the unique carving decomposition of $G$ when $|V(G)| = 3$
14. while $\text{contractionStack}$ is not empty do
15. pop $\langle \langle u, v \rangle, t \rangle$ from $\text{contractionStack}$
16. add two new leaves $L_u$ and $L_v$ to $\text{decomposition}$
17. make the leaf $L_t$ in $\text{decomposition}$ parent of $L_u$ and $L_v$
18. end while
19. return $\text{decomposition}$

### 4.4.2 Running Time Analysis

From the pseudo-code of algorithm 4.3, the first step takes $O(|V|^2 \cdot \log |V|)$ time due to the analysis in the previous section. The while loop has exactly $|V| - 3$ iterations. During each iteration, each edge in $E(G)$ is examined by calling $\text{ST-Procedure}$ to
find an edge $e$ such that $G \setminus e$ has carving width at most $k$. Therefore, there are at most $|E|$ calls to ST-PROCEDURE during each iteration. The running time of other steps are small or dominated by the running time of ST-PROCEDURE. Hence, the overall running time of algorithm 4.3 is $O(|E| \cdot |V|^3) = O(|V|^4)$. 
Chapter 5

Implementations

The code implemented in this thesis project builds upon a set of C++ classes developed by Dr. David Juedes for CS 604 in 2006. Here, we extend the class hierarchy by adding two more types of decompositions. The code for these classes uses the Public Implementation of a Graph Library and Editor (PIGALE) [1] as a supporting library for the whole package. The other classes were provided by Dr. Juedes as a foundation for writing efficient and practical implementations of theoretical graph algorithms.

5.1 Software Overview

The software mainly consists of classes dealing with different type of graph decompositions. There are five classes for decompositions: Series Parallel Decomposition, Tree Decomposition, Nice Tree Decomposition, Branch Decomposition, and Carving Decomposition. The contribution of this thesis are the two graph decomposition classes: Branch Decomposition and Carving Decomposition. These classes derive from the template class Tree\langle T\rangle which supports several common tree operations like insert subtree, insert a new child, find degree of the tree, find the size of the tree, tree traversal. Class Tree\langle T\rangle implements the first-
child right-sibling tree structure that is flexible enough to handle different types of decompositions.

The software also includes a simple BaseGraph that describes a graph by its adjacency list. We also use the PIGALE library to process planarity related operations. Our BaseGraph can convert itself to objects of types TopologicalGraph and GraphContainer, the graph structural classes provided by PIGALE. These conversions are useful in Branch Decomposition and Carving Decomposition when we need to check for planarity and construct medial graphs or dual graphs.

<table>
<thead>
<tr>
<th>Class</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseGraph</td>
<td>Keeps the structure of a graph, and is normally used as input to the decomposition classes.</td>
</tr>
<tr>
<td>Tree(T)</td>
<td>Supports common tree operations, implements left-child right-sibling structure.</td>
</tr>
<tr>
<td>Series Parallel Decomposition</td>
<td>Checks if a given graph in a BaseGraph object is series-parallel and constructs a series-parallel decomposition if it is.</td>
</tr>
<tr>
<td>Tree Decomposition</td>
<td>Constructs an optimal tree decomposition from a series-parallel decomposition of a given graph in a BaseGraph object.</td>
</tr>
<tr>
<td>Nice Tree Decomposition</td>
<td>Constructs a nice tree decomposition from a tree decomposition of a given graph in a BaseGraph object.</td>
</tr>
<tr>
<td>Carving Decomposition</td>
<td>Checks if a given graph in either a BaseGraph object or a TopologicalGraph object is a planar graph, and constructs an optimal carving decomposition if it is.</td>
</tr>
<tr>
<td>Branch Decomposition</td>
<td>Checks if a given graph in either a BaseGraph object or a TopologicalGraph object is a planar graph, and constructs an optimal rooted branch decomposition if it is. Also converts itself to a tree decomposition.</td>
</tr>
</tbody>
</table>
5.2 Implementations

This thesis focuses on the implementations of two classes Branch Decomposition and Carving Decomposition. As mentioned in the previous chapter, finding an optimal branch decomposition for planar graphs essentially based on finding an optimal carving decomposition of the medial graphs. The ST-PROCEDURE (algorithm 4.1) is implemented as a protected member function of the Carving Decomposition class. The main function build\_decomposition implements algorithm 4.3 by invoking ST-PROCEDURE $O(n^2)$ times. Branch Decomposition uses Carving Decomposition to construct an optimal carving decomposition and then converts it to a branch decomposition. Branch Decomposition also provides a function to convert branch decomposition to tree decomposition following the steps in section 3.4.

On the other hand, pre-conditions of any function in these two classes is to make sure the input graph is planar and to have an embedding of the given graph. Therefore, our implementations use P.I.G.A.L.E, an open-source library, to verify planarity of the input graph.

5.2.1 The PIGALE Library

PIGALE is an open source package developed by Fraysseix and Mendez [1] mainly for various operations on planar graphs. The package is intended for implementations of algorithms in graph theoretical research. It consists of three components: tgraph - a core library that implements graph algorithms, PIGALE - a graph editor with
graphical interface to invoke the core library functionalities, and client - a client with the ability to interact with the graph editor over the network. For the purpose of our implementation, we only use the core library tgraph in the PIGALE package. More specifically, we take advantage of tgraph’s data structure for planar graphs, the function to check for planarity, and the function to build a dual graph.

We use two classes from PIGALE are TopologicalGraph and GraphContainer. GraphContainer keeps the main structure of a graph, i.e., its vertices and edges. TopologicalGraph has a protected object of type GraphContainer to maintain the graph structure. It also provides member functions to do operations on the graph stored in its GraphContainer object such as checking for planarity, checking if the graph is connected, marking vertices according to their connected components, checking for 2-connectivity, getting the minimum and maximum degrees of vertices in the graph, contracting edges, etc.

The data structures in GraphContainer and TopologicalGraph also allows developers to access vertices and edges belonging to the same face in constant time. It also provides random access to edges incident to a given vertex in counter-clockwise or clockwise order in its embedding. In particular, each edge in a graph with $m$ edges is given an index $1 \leq i \leq m$. A brin is a half-edge corresponding to an edge and one of its end points. Brins are given indices from $-m$ to -1 and 1 to $m$. Each vertex is assigned an index from 1 to $n$. The internal structure uses set of brins ($B$), set of vertices ($V$), and set of edges ($E$) to make graph traversals efficient [32].
Implementation-wise, PIGALE overloads the \([\ ]\) operator and provides the following mapping between these three sets.

Table 5.2: Mapping between brins \((B)\), vertices \((V)\), and edges \((E)\) [32]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e)</td>
<td>(E \rightarrow B)</td>
<td>the first brin of edge (e)</td>
</tr>
<tr>
<td>(-b)</td>
<td>(B \rightarrow B)</td>
<td>brin opposite to brin (b)</td>
</tr>
<tr>
<td>(</td>
<td>b</td>
<td>)</td>
</tr>
<tr>
<td>(vin[b])</td>
<td>(B \rightarrow V)</td>
<td>vertex of the half-edge (b)</td>
</tr>
<tr>
<td>(pbrin[v])</td>
<td>(V \rightarrow B)</td>
<td>first brin incident to vertex (v)</td>
</tr>
<tr>
<td>(cir[b])</td>
<td>(B \rightarrow B)</td>
<td>brin next to brin (b) in circular order</td>
</tr>
<tr>
<td>(acir[b])</td>
<td>(B \rightarrow B)</td>
<td>brin before brin (b) in circular order</td>
</tr>
</tbody>
</table>

Moreover, their flexible data structure allows developers to dynamically create and assign properties to vertices and edges of the graph. For example, the developer can create a property for the name of each vertex and a property for the weight of each edge.

One noticeable difference between PIGALE and our implementation is that all indices are 1-based in PIGALE while we use 0-based indexing scheme. Therefore, we have a step to convert the vertex indices from one type to another at the beginning and at the end of some procedures.

### 5.2.2 Our Data Structures

Our internal graph structure is stored in the `BaseGraph` object. It simply keeps the adjacency list of the graph. Moreover, it also has private variables to store a `GraphContainer` object and `TopologicalGraph` object corresponds to the adjacency list. These objects help `Branch Decomposition` and `Carving Decomposition`
get the planar embedding of the graph directly with the accessors we provided in BaseGraph.

5.2.3 ST-Procedure

The ST-Procedure is implemented as a protected member function of the Carving Decomposition class. Because we only call this function internally from the build_decomposition function, it takes a TopologicalGraph $G$ object and an integer $k$ as input. The function will return TRUE if the given graph has carving-width $< k$ or return FALSE otherwise. The Carving Decomposition class also has two private member variables $D$ to store the all-pair-shortest-path matrix and a member variable $Q$ to used as a queue when necessary. We do not use STL queue class to prevent unnecessary computation overhead due to their templated implementation.

The ST function implements exactly what described in algorithm 4.1. At the first step, we find the maximum degree of vertices in the given graph using a PIGALE API and compare it with $k$. We then use the PIGALE API DualGraph to construct $dG$, the dual graph of $G$. Then we run breadth-first-search on each and every vertex of $dG$ to find all-pair-shortest-paths and store it in the class variable $D$.

Listing 5.1: Declarations of ncc and mark

```cpp
Prop<int> ncc(G.Set(tedge()), PROP_RESERVED + 1);
    // number of connected components of G_v for each edge
Prop<svector<int>> mark(G.Set(tedge()), PROP_RESERVED + 2);
    // mark each vertex belongs to which CC of G_v for each edge
```
For the game state table $RC$, we take advantage of the dynamically created properties of PIGALE. First of all, we create an integer property named $ncc$ for each edge $e$ in $G$ to represent the number of connected components in $G_e$. To provide better access to each connected components in $G_e$, we also have an array (of size $|V(G)|$) property named $mark$ for each edge $e$ in $G$. That means for each edge $e$, $mark[e]$ is an array marking each and every vertex of $G$ with the index of the connected components of $G_e$ it belongs to.

To represent the game state table $RC$, we create an array as a property named $MarkXe$ for each edge in $G$. The array for each edge $e$ is a boolean array of size $ncc[e]$ represents whether a game state $RC(e, i)$ with $e \in E(G)$ and $1 \leq i \leq C$ is marked as losing state where $C$ is the number of components in $G_e$. Similarly, we create an array as a property named $MarkXr$ for each vertex in $dG$. Notice that each vertex in the dual graph $dG$ corresponds to one and only one face in the original graph $G$, hence a property for each vertex in $dG$ means a property for each face in $G$. Therefore, the property $MarkXr$ represents whether a game state $RC(r, v)$ with $r \in R(G) = V(dG)$ and $v \in V(G)$ is marked as a losing state.
Listing 5.3: Declarations of CountXe and CountXr

```cpp
Prop<int> CountXe(G.Set(tedge()), PROP_RESERVED + 3);
    // count # of elements for Xe
Prop<int> CountXr(dG.Set(tvertex()), PROP_RESERVED + 5);
    // count # of elements for Xr (or Xv for dG)
```

To monitor whether all states related to an edge or a face are marked as losing states, we have a counter for each edge of \( G \) and each face of \( G \) (i.e., each vertex of \( dG \)). The counters are integer properties named \texttt{CountXe} for edges of \( G \) and \texttt{CountXr} for vertices of \( dG \).

Listing 5.4: Declaration of CountRE

```cpp
Prop<svector<int>> CountRE(G.Set(tbrin()), PROP_RESERVED + 4);
    // count # of elements for \((e, r, C)\)
    // since there are 2 r’s for each e, each of them corresponds
    // to 1 brin of edge e (−e and e)
```

Finally, according to the algorithm, we also need a counter for each pair \((e, r, C)\) to keep track of number of losing states adjacent to \((e, C)\) from face \( r \) incident to \( e \). Since there are exactly two faces adjacent to \( e \) each of which can be mapped to one half-edge (brin) of \( e \), we create an array (of size \texttt{ncc}[e]) property named \texttt{CountRE} for each brin of \( G \). For each brin \( b \) of \( G \) \((-m \leq b \leq -1 \text{ or } 1 \leq b \leq m)\), \texttt{CountRE}[b] is an array where \texttt{CountRE}[b][i] represents the number of unmarked states adjacent to \((|b|, i)\) from face \( r \) corresponding to brin \( b \) of \( e \).

The processing stack in algorithm 4.1 is declared as a stack of \texttt{pair<int, int>} named \texttt{L}. We use a simple coding method to represent each game state with a pair
of integers \( \langle x, y \rangle \). If \( x > |E(G)| \), then \( \langle x, y \rangle \) represents a game state \((r, v)\) for a face \( r \) in \( G \) and a vertex \( v \) in \( V(G) \) where \( r = x - |E(G)| \) and \( v = y \). Otherwise, \( \langle x, y \rangle \) represents a game state \((e, i)\) for an edge \( e \) in \( G \) and an integer value \( 1 \leq i \leq ncc[e] \) where \( e = x \) and \( i = y \).

After all the data structures are created as described above, the code works exactly like the pseudo-code of algorithm 4.1. At the end of the function or before any \texttt{return} statement (due to the fact that any of the \texttt{CountXe} or \texttt{CountXr} value reaches 0), we make a call to the function \texttt{clearCustomizedGraph} to remove all the dynamically created properties that we setup at the beginning of the function. This step is executed with the intention of preventing memory leaks caused by property objects not being removed from the \texttt{TopologicalGraph} object.

### 5.2.4 Building An Optimal Carving Decomposition

The function \texttt{build_decomposition} is the most important function provided in the \texttt{Carving Decomposition} class. It applies binary search using the bound mentioned in section 4.3 to find the value of carving-width for the given graph. Moreover, the all-pair-shortest-path array \( D \) is dynamically created at the beginning of this function, and will be re-used in any call to the \texttt{ST} function later on. It is finally deallocated at the end of this function.

Algorithm 4.3 is implemented easily with the use of PIGALE API \texttt{ContractEdge}. This API will contract a given edge \( e \) and return index of the new vertex in the
contracted graph. After the whole construction algorithm is executed, we have a binary tree of an optimal carving decomposition. This tree satisfies all condition of a carving decomposition except for the root node. This is the intention of our implementation because we can easily convert it to a regular carving decomposition by removing the root node and make its two children adjacent to each other.

Since the purpose of building a decomposition is to use it in dynamic programming algorithms, we also construct the midset of each node in the tree at the end of the function.

Listing 5.5: Definition of a node in carving decompositions

```cpp
struct CarvingNodeInfo {
    size_t u;  // index of the vertex for a leaf node
    set<size_t> infoSet;  // vertex set or midset of a node
};
```

Our structure of the node in the resulting tree only has two fields: an integer $v$ to indicate which vertex in $G$ a leaf in the decomposition corresponds to, and a set of integer named `infoSet`. For non-leaf node, the value of $v$ is 0 (remember that we are dealing with PIGALE indexing scheme which is 1-based). We construct the midsets using a 2-stage process. The first stage, we use the field `infoSet[i]` for each node $i$ in the carving decomposition to store the set of vertex indices in the subtree rooted at $i$. In the second stage, we for each `infoSet[i]`, we form a new set of edge indices connecting a vertex in `infoSet[i]` to a vertex in $V \setminus \text{infoSet}[i]$. This new set is the midset of $(i, p[i])$ by definition 4.6. Finally, we assign it back to
infoSet \[i\]. After this 2-stage process, the value of infoSet \[i\] is the midset of the edge connecting \(i\) to its parent in the carving decomposition tree.

5.2.5 Building An Optimal Branch Decomposition

The class Branch Decomposition heavily depends on Carving Decomposition to construct an optimal branch decomposition of a given graph. It first checks whether the input graph \(G\) is planar, and then uses our helper function to construct the medial graph \(mG\) of \(G\). After that, a Carving Decomposition object is created to construct the carving decomposition of \(mG\). The structure of each node in a branch decomposition is defined as follows:

Listing 5.6: Definition of a node in branch decompositions

```c++
struct BranchNodeInfo {
    size_t u, v; // endpoints of the edge for a leaf node
    set<size_t> infoSet; // edge set or midset of a node
};
```

A simple conversion from the carving decomposition structure back to the branch decomposition structure follows. After that step, we have a branch decomposition where each non-leaf node has an empty infoSet and each leaf node stores the edge index instead of the edge endpoints. In a fashion similar to the 2-stage process of building midsets for a carving decomposition, we also build the midsets for the current branch decomposition with a 2-stage process. First we use infoSet \([i]\) to store the edge set of the sub-tree rooted at \(i\) recursively. Then for each node \(i\) in
the branch decomposition, we convert infoSet \([i]\) from the edge set to the midset of that node. After the midsets are constructed, we convert the edge index for each leaf into a pair of vertex indices of its endpoints and store them in \(u\) and \(v\) in the structure BranchNodeInfo. Finally, we do a traversal in the branch decomposition to change all the indices from 1-based to 0-based.

\[
\begin{array}{cccc}
1 & 4 & 6 & 8 & 2 & 5 & 7 & 11 & 9 & 20 & 16 & 13 \end{array}
\]

nccIdx array

\[
\begin{array}{c}
nccCount[1] = 5 \quad nccCount[2] = 4 \quad nccCount[t] = 2 \end{array}
\]

Figure 5.1: Vertices of the same connected components in continuous blocks

Moreover, we also need an efficient way of traveling in each connected components of \(G_e\) when processing a state \(RC(e, i)\) (algorithm 4.1). We created three array properties \(nccCount\), \(nccIdxStart\), and \(nccIdx\) to arrange the vertices in the same connected components of \(G_e\) into continuous blocks of a size-|\(V\)| array. By doing this, checking all elements of a connected components with \(C\) vertices takes exactly |\(C\)| steps.

### 5.2.6 Some Improvements

Due to the number of arrays used for marking and counting in the implementations, we suffered from insufficient memory issues during our first trial. The implemen-
tation was improved by the following heuristics in an attempt to reduce unnecessary memory usages:

**Improvement A.** The additional arrays of nccCount, nccIdxStart, and nccIdx become useless when $G_e$ has exactly $|V|$ connected components, i.e., each vertex is on a separate component. This case happens when the ST-PROCEDURE is called with a large value of $k$. Therefore, we check if ncc[e] equals to $|V|$ for each edge $e \in E$. If that is the case, we will not create the properties nccCount, nccIdxStart, and nccIdx for $e$.

**Improvement B.** Similar to improvement A, the property array CountRE also becomes useless when $G_e$ has exactly $|V|$ connected components. In that case, CountRE[b][i] is exactly 1 for every $i$ from 1 to $V$ ($b$ is a brin with value $e$ and $-e$). Therefore, we can remove this property for such $e$.

To reduce the number of calls to ST-PROCEDURE, we use a simple heuristic in the step for finding an edge to contract.

**Improvement C.** We keep a global index de of the edge contracted from the previous iteration. Since the edge de is contracted right before the current iteration, it means that edges with index lower than de probably failed the ST-PROCEDURE test for contracting edges. Therefore, we start the current iteration from de to prevent repeated failures.
5.3 Experimental Results

We test our implementation on three different set of graphs: grid graphs, \( r \)-outer planar graphs, and series-parallel graphs. The results are shown in tables 5.3, 5.4, and 5.5.

We measure the running time and number of calls to the ST-PROCEDURE for two phases: the decision phase and the construction phase. The decision phase happens when algorithm 4.2 is executed, i.e., finding the carving-width for the input graph. This step uses binary search on the upper bounds of carving-width for planar graphs by asking the ST-PROCEDURE if a given value is greater than the carving-width.

The construction phase is the main part of algorithm 4.3, i.e., it repeatedly contracts a feasible edge until the reduced graph has at most three vertices. This phase is the most costly step and its running time normally dominates the running time of all other steps. There are also a few other steps after the construction phase such as: converting the optimal carving decomposition to an optimal branch decomposition, building the midsets, fixing the indices in the resulting tree. Results from the experiment shows that these steps take very little amount of time comparing to the decision phase and the construction phase. Therefore, we only monitor the decision and construction phases.

The input cases were tested on a computer with 2 x 2.16 Ghz processors, 2 GBytes of physical memory among which 1.5 GBytes was available for the software. Notice that on the result table, the number of vertices and edges are for the input graph
Table 5.3: Results for test set of grid graphs

<table>
<thead>
<tr>
<th>Test</th>
<th>$V$</th>
<th>$E$</th>
<th>$\beta(G)$</th>
<th>Decision phase</th>
<th>Construction phase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST calls</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time</td>
<td>ST calls</td>
</tr>
<tr>
<td>GRID01</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0.000</td>
</tr>
<tr>
<td>GRID02</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.000</td>
</tr>
<tr>
<td>GRID03</td>
<td>10</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td>0.016</td>
</tr>
<tr>
<td>GRID04</td>
<td>20</td>
<td>19</td>
<td>2</td>
<td>3</td>
<td>0.016</td>
</tr>
<tr>
<td>GRID05</td>
<td>8</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
</tr>
<tr>
<td>GRID06</td>
<td>16</td>
<td>22</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
</tr>
<tr>
<td>GRID07</td>
<td>20</td>
<td>28</td>
<td>2</td>
<td>4</td>
<td>0.000</td>
</tr>
<tr>
<td>GRID08</td>
<td>40</td>
<td>58</td>
<td>2</td>
<td>4</td>
<td>0.031</td>
</tr>
<tr>
<td>GRID09</td>
<td>30</td>
<td>47</td>
<td>3</td>
<td>4</td>
<td>0.031</td>
</tr>
<tr>
<td>GRID10</td>
<td>60</td>
<td>97</td>
<td>3</td>
<td>5</td>
<td>0.063</td>
</tr>
<tr>
<td>GRID11</td>
<td>90</td>
<td>147</td>
<td>3</td>
<td>5</td>
<td>0.125</td>
</tr>
<tr>
<td>GRID12</td>
<td>40</td>
<td>66</td>
<td>4</td>
<td>4</td>
<td>0.031</td>
</tr>
<tr>
<td>GRID13</td>
<td>80</td>
<td>136</td>
<td>4</td>
<td>4</td>
<td>0.094</td>
</tr>
<tr>
<td>GRID14</td>
<td>120</td>
<td>206</td>
<td>4</td>
<td>5</td>
<td>0.265</td>
</tr>
<tr>
<td>GRID15</td>
<td>200</td>
<td>346</td>
<td>4</td>
<td>5</td>
<td>0.656</td>
</tr>
<tr>
<td>GRID16</td>
<td>125</td>
<td>220</td>
<td>5</td>
<td>5</td>
<td>0.562</td>
</tr>
<tr>
<td>GRID17</td>
<td>250</td>
<td>445</td>
<td>5</td>
<td>6</td>
<td>1.281</td>
</tr>
<tr>
<td>GRID18</td>
<td>500</td>
<td>895</td>
<td>5</td>
<td>6</td>
<td>8.469</td>
</tr>
<tr>
<td>GRID19</td>
<td>500</td>
<td>895</td>
<td>5</td>
<td>6</td>
<td>7.468</td>
</tr>
<tr>
<td>GRID20</td>
<td>250</td>
<td>445</td>
<td>5</td>
<td>6</td>
<td>1.281</td>
</tr>
<tr>
<td>GRID21</td>
<td>125</td>
<td>220</td>
<td>5</td>
<td>5</td>
<td>0.438</td>
</tr>
<tr>
<td>GRID22</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>0.016</td>
</tr>
<tr>
<td>GRID23</td>
<td>16</td>
<td>24</td>
<td>4</td>
<td>3</td>
<td>0.000</td>
</tr>
<tr>
<td>GRID24</td>
<td>36</td>
<td>60</td>
<td>6</td>
<td>4</td>
<td>0.031</td>
</tr>
<tr>
<td>GRID25</td>
<td>100</td>
<td>180</td>
<td>10</td>
<td>5</td>
<td>0.391</td>
</tr>
<tr>
<td>GRID26</td>
<td>400</td>
<td>760</td>
<td>20</td>
<td>6</td>
<td>7.969</td>
</tr>
<tr>
<td>GRID27</td>
<td>900</td>
<td>1740</td>
<td>30</td>
<td>7</td>
<td>28.157</td>
</tr>
<tr>
<td>GRID28</td>
<td>1600</td>
<td>3120</td>
<td>40</td>
<td>7</td>
<td>90.594</td>
</tr>
<tr>
<td>GRID29</td>
<td>2500</td>
<td>4900</td>
<td>50</td>
<td>8</td>
<td>316.141</td>
</tr>
<tr>
<td>GRID30</td>
<td>1250</td>
<td>2425</td>
<td>25</td>
<td>7</td>
<td>74.532</td>
</tr>
<tr>
<td>Test</td>
<td>V</td>
<td>E</td>
<td>β(G)</td>
<td>Decision phase</td>
<td>Construction phase</td>
</tr>
<tr>
<td>-------</td>
<td>----</td>
<td>----</td>
<td>------</td>
<td>----------------</td>
<td>--------------------</td>
</tr>
<tr>
<td></td>
<td>ST calls</td>
<td>Time</td>
<td>ST calls</td>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>OUTER01</td>
<td>14</td>
<td>25</td>
<td>2</td>
<td>4</td>
<td>0.000</td>
</tr>
<tr>
<td>OUTER02</td>
<td>28</td>
<td>53</td>
<td>2</td>
<td>4</td>
<td>0.062</td>
</tr>
<tr>
<td>OUTER03</td>
<td>50</td>
<td>97</td>
<td>2</td>
<td>5</td>
<td>0.078</td>
</tr>
<tr>
<td>OUTER04</td>
<td>75</td>
<td>170</td>
<td>3</td>
<td>5</td>
<td>0.156</td>
</tr>
<tr>
<td>OUTER05</td>
<td>150</td>
<td>345</td>
<td>3</td>
<td>6</td>
<td>0.875</td>
</tr>
<tr>
<td>OUTER06</td>
<td>225</td>
<td>520</td>
<td>3</td>
<td>6</td>
<td>2.187</td>
</tr>
<tr>
<td>OUTER07</td>
<td>100</td>
<td>243</td>
<td>4</td>
<td>5</td>
<td>0.390</td>
</tr>
<tr>
<td>OUTER08</td>
<td>200</td>
<td>493</td>
<td>4</td>
<td>5</td>
<td>1.547</td>
</tr>
<tr>
<td>OUTER09</td>
<td>400</td>
<td>993</td>
<td>4</td>
<td>6</td>
<td>6.921</td>
</tr>
<tr>
<td>OUTER10</td>
<td>125</td>
<td>316</td>
<td>5</td>
<td>5</td>
<td>0.672</td>
</tr>
<tr>
<td>OUTER11</td>
<td>250</td>
<td>641</td>
<td>5</td>
<td>6</td>
<td>4.031</td>
</tr>
<tr>
<td>OUTER12</td>
<td>500</td>
<td>1291</td>
<td>5</td>
<td>6</td>
<td>9.015</td>
</tr>
<tr>
<td>OUTER13</td>
<td>100</td>
<td>261</td>
<td>10</td>
<td>5</td>
<td>0.891</td>
</tr>
<tr>
<td>OUTER14</td>
<td>400</td>
<td>1121</td>
<td>20</td>
<td>7</td>
<td>7.843</td>
</tr>
<tr>
<td>OUTER15</td>
<td>900</td>
<td>2581</td>
<td>30</td>
<td>7</td>
<td>61.688</td>
</tr>
</tbody>
</table>

The results suggest that our implementation of algorithm 4.3 can handle planar graphs with up to 2000 edges. For bigger graphs (test cases GRID28-30, OUTER15), we can not construct optimal carving decompositions in a reasonable amount of time. Without improvement A and B, our implementation failed on graphs with more than 1200 edges (test cases GRID27-28, OUTER12).

However, the decision phase can still finish relatively fast for most cases. Improvements A and B also help our implementation find the carving-width for graphs with large number of edges (test cases GRID29-30, OUTER15). The slowest running

$G$ while ST-PROCEDURE are called on the medial graph $G_m$ of $G$. Therefore, the running time of $O(n^4)$ and $O(n^2)$ for algorithms 4.3 and 4.1, respectively, are actually measured with $n = |E(G)|$. 
Table 5.5: Results for test set of series-parallel graphs

<table>
<thead>
<tr>
<th>Test</th>
<th>$V$</th>
<th>$E$</th>
<th>$\beta(G)$</th>
<th>Decision phase</th>
<th>Construction phase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST calls</td>
<td>ST calls</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td>SP01</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>SP02</td>
<td>9</td>
<td>13</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>SP03</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP04</td>
<td>14</td>
<td>21</td>
<td>2</td>
<td>3</td>
<td>0.016</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>SP05</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>SP06</td>
<td>17</td>
<td>24</td>
<td>2</td>
<td>4</td>
<td>0.016</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>SP07</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>SP08</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>SP09</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>SP10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finally, from the result tables, the number of calls to ST-PROCEDURE is at most twelve times the number of vertices in the medial graph, i.e., the number of edges in the input. This observation suggests that improvement C helps in reducing the number of ST-PROCEDURE calls.
Chapter 6

Conclusions and Future Work

This thesis studies and explores algorithms based on graph decompositions. More specifically, dynamic programming algorithms based on tree decompositions, path decompositions, and branch decompositions for Independent-Set are given as examples of how hard problems can be solved efficiently on graphs with decompositions of small width.

The main contribution of this thesis is a stable implementation of the constructive algorithm for computing optimal branch decompositions for planar graphs. Our purpose is to create a stable and efficient code base of implementations for practical graph algorithms. We contributed to the existing code base two additional types of decompositions: carving decomposition and branch decomposition. Our implementation is able to construct an optimal branch decomposition for planar graphs with up to 2000 edges and decide the branch-width for planar graphs with (at least) up to 4900 edges.

A secondary contribution of this thesis is the extensive explanation of concepts and proofs related to branch decompositions, carving decompositions, medial and dual graphs, and the rat-catching game.

With a stable implementation for constructing optimal branch decompositions for planar graphs in hand, we can combine it with dynamic programming on subsets with
a branch-width based approach (e.g., algorithm 3.5 from section 3.3.2) to solve NP-HARD problems on planar graphs efficiently. This leads to the algorithm framework 6.1 for solving hard problems restricted to planar graphs.

**Algorithm 6.1** Solving hard problems on planar graphs

1. check for planarity of $G$. If $G$ is not planar, stop
2. use algorithm 4.3 to construct an optimal branch decomposition for $G$
3. apply a dynamic programming on subset on the constructed decomposition to solve the original problem in $O(c^{\beta(G)} \cdot p(n))$ time.

Finally, our aim is also to attack bio-informatics problems using optimal branch decompositions. More specifically, we want provide better algorithms for some RNA related problems [22, 23, 30, 31] that are known to have fast solutions with the tree decomposition approach. For example, we might be able to use optimal branch decompositions to improve the work by Song et al. [30] for the RNA alignment problem [30] in 2006. In their solution, the first step was to remove some edges from the input RNA graph to make it planar. Since RNA graphs are quite close to planar graphs, the number of edges removed from the original graph was not big. In addition to the resulting graph being planar, the graph also had a simple structure that was exploited via a simple heuristic algorithm to construct a tree decomposition of small width. After this step, the algorithm restored all the previously removed edges to create a tree decomposition for the original RNA graph. Finally, a dynamic programming algorithm was applied on this tree decomposition to solve the RNA alignment problem.
Further research will need to be conducted to determine whether using a branch decomposition approach in this case will lead to faster algorithms. The reason behind this idea is that an optimal branch decomposition can be constructed efficiently for planar graphs, therefore it may defeat the tree decomposition in their algorithm if its approximated width is not close to the actual tree-width. Since the width of the given decomposition is the exponent of the exponential factor in the running time, it is a significant factor in how fast the algorithm can be. Consequently, a slight decrement of the width can potentially speed up the algorithm a lot. Therefore, using an optimal branch decomposition approach may provide a faster algorithm.
Bibliography


Springer-Verlag.


[15] Fedor V. Fomin and Dimitrios M. Thilikos. New upper bounds on the decom-
posability of planar graphs and fixed parameter algorithms. Technical report,
Universitat Politcnica de Catalunya, 2002.


[30] Yinglei Song, Chunmei Liu, Xiuzhen Huang, Russell L. Malmberg, Ying Xu, and Liming Cai. Efficient parameterized algorithms for biopolymer structure-


Appendix A

Source Code

A.1 Class Carving_Decomposition

Listing A.1: Carving_Decomposition.h

```cpp
// Carving_Decomposition.h
FILE: Carving_Decomposition.cc
AUTHOR: Hiep Dinh
DATE: July 2008
DESCRIPTION: Carving_Decomposition builds an optimal carving decomposition for a given planar graph.
the ST_Procedure is implemented as a private function in this class and is used to construct the carving decomposition.

#ifndef CARVING_DECOMPOSITION_H
#define CARVING_DECOMPOSITION_H

#include "GraphUtils.h"
#include "Tree.h"
#include "BaseGraph.h"
#include <Pigale.h>
#include <TAXI/netcut.h>
#include <map>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cassert>
#include <vector>
#include <set>
#include <stack>
#include <list>
using namespace std;

struct CarvingNodeInfo {
    size_t u;
    set<size_t> infoSet;
};

class Carving_Decomposition : public Tree<CarvingNodeInfo> {

public:
    bool build_decomposition(BaseGraph &G);
    bool build_decomposition(TopologicalGraph* TG);
    int width() {
        return rec_width(root_ptr());
    }

    void print() {

    }

};
```
int counter = -1;
rec_print(root_ptr(), counter);
}

Carving_Deepomposition::Carving_Deepomposition() {
is_formed = false;
Q = NULL;
G = NULL;
D = NULL;
}

protected:
bool findCarvingDecomposition (int upperBound);
bool ST(TopologicalGraph & G, int k);
void formInfoVSet (iterator p);
void formInfoMidSet (iterator p);

// helper functions
int GenerateGe (TopologicalGraph & G, tedge e, int k, short int ** D, svector<short int> & mark);
void clearCustomizedGraph (TopologicalGraph & G, TopologicalGraph & dG, GraphContainer * dGC);

// helper variables
short int ** D;
short int * Q;
private:
TopologicalGraph * G;
int STCalls;
int lastSTCounter;
bool firstCalled;
clock_t lastTime;
void rec_time(string status = "Check time");
bool is_formed;
void rec_print(iterator p, int & counter) {
iterator p2;
list<int> sub_tree_nums;
list<int>::iterator p1;
set<size_t>::iterator p3;
if (p == end()) {
return;
}
for (p2 = first_child(p); p2 != end(); ++p2) {
rec_print(p2, counter);
sub_tree_nums.push_back(counter);
}
counter++;
if (sub_tree_nums.empty()) {
cout << counter << "L"u << (*p).u << "*";
for (p3 = (*p).infoSet.begin(); p3 != (*p).infoSet.end(); ++p3) {
cout << "u"u << *p3;
}
cout << "u"u << endl;
} else {
cout << counter << "I"u << (*p).u << "*";
for (p3 = (*p).infoSet.begin(); p3 != (*p).infoSet.end(); ++p3) {
cout << "u"u << *p3;
}
cout << "u"u << *p1;
for (sub_tree_nums.begin(); p1 != sub_tree_nums.end(); ++p1) {
```cpp
int rec_width(Carving_Decomposition::iterator p) {
    int max, temp;
    Carving_Decomposition::iterator p1;
    if (p != end()) {
        max = (*p).infoSet.size();
        for (p1 = first_child(p); p1 != end(); ++p1) {
            temp = rec_width(p1);
            if (max < temp) max = temp;
        }
    }
    return max;
}
return 0;
}
```
// find an optimal carving decomposition for the graph in this G
// using upperBound to speed up the process of finding the carving-width value
bool Carving_Decomposition::findCarvingDecomposition (int upperBound) {
    // initialize timer
    firstCalled = true;
    STCalls = 0;
    lastSTCounter = 0;
    currentTime();

    // initialize the arrays used in each set
    D = new int [G->ne() + 1];
    for (int i = 1; i <= G->ne(); ++i) D[i] = new int [G->ne() + 1];
    Q = new int [G->ne() + 1];

    // find the smallest k that fits, should be an even number
    int k = 1;
    int dmin, dmax;
    G->MinMaxDegree (dmin, dmax);
    int lo = dmax / 2;
    int hi = upperBound;

    // binary search
    while (lo < hi) {
        int mid = (lo + hi) / 2;
        if (ST(*G, 2 * mid + 1))
            hi = mid;
        else
            lo = mid + 1;
    }

    checkTime("Found the carving width in");
    // we use k = carvingWidth + 1 because of how the ST-Procedure
    // collects the input
    k = lo * 2 + 1;

    // the carving-width is k - 1
    int V, E;
    GraphContainer* cGC = new GraphContainer (G->Container());
    TopologicalGraph cG (*cGC);
    int iter = 0;

    // the info array is used to store the edge contraction list
    // for the i-th contraction, an edge between vertices info [1][i]
    // and info [2][i] is contracted to a new vertex info [0][i]
    int* info[3];

    // initialize the info array
    for (int i = 0; i < 3; ++i) {
        info[i] = new int [2 * cG.nv()];
        memset (info[i], 0, 2 * cG.nv() * sizeof(int));
    }

    Prop<int> originalValue(cG.Set(tvertex())), PROP_RESERVED + 8);
    for (tvertex v = 1; v <= cG.nv(); ++v) originalValue[v] = v();
    iter = cG.nv();
}
int e = 1;
ext int total = 0;
// iteratively find the edge contraction order
int found = 1;
while ((cG.nv() > 3 && found) {
  ++iter;
  E = cG.ne();
  V = cG.nv();

  found = 0;
  Prop<int> realValue(cG.Set(tvertex()), PROP_RESERVED + 8);
  // the tested map is used to prevent trying to contract edges
  // between the same pair of vertices because in our graph, there
  // may be multiple edges connecting the same vertices
  map<int, set<int>> tested;
  // the index e is continued from the previous loop
  // this heuristic helps us skip all the failed tested edges in
  // the previous round, and only try it again if the rest of
  // the edge set has all failed
  for (int de = 1; de <= E && !found; ++de, ++e) {
    int ou = realValue[cG.vin[e]], ov = realValue[cG.vin[-e]];
    if (ou == ov) continue;
    if (ou > ov) swap(ou, ov);
    // check if an edge between ou & ov has been tested before
    if (tested.find(ou) != tested.end()) continue;
    tested.insert(ov);
    // create a new copy of the current graph, prepare to contract
    GraphContainer* ccGC = new GraphContainer(cG.Container());
    TopologicalGraph ccG(*ccGC);
    tvertex nv = ccG.ContractEdge(e);

    if (!ccG.FindPlanarMap()) {
      ccG.Set().erase(PROP_RESERVED + 8);
      ccG.reset();
      delete ccGC;
      continue;
    }
    // if call ST-procedure to see if e is the edge we can contract
    if (ST(ccG, k)) {
      // found the edge
      found = 1;
      nv = cG.ContractEdge(e);
      total += de;
      Prop<int> updatedValue(cG.Set(tvertex()), PROP_RESERVED + 8);
      updatedValue[nv] = iter;
      info[0][ou] = info[0][ov] = iter;
      info[1][iter] = ou;
      info[2][iter] = ov;
    }
    ccG.Set().erase(PROP_RESERVED + 8);
    ccG.reset();
    delete ccGC;
  }
}
if (found) {
  // build the decomposition
  E = cG.ne();
  V = cG.nv();
}
Prop<int> realValue(cG.Set(tvertex())); // we have 3 vertices left, manually form binary tree from // these last pair of info
++iter;

info[0][realValue[1]] = info[0][realValue[2]] = iter;
info[1][iter] = realValue[1];
info[2][iter] = realValue[2];
if (info[1][iter] > info[2][iter]) swap(info[1][iter], info[2][iter]);

if (cG.nv() > 2) {
    ++iter;
    info[0][realValue[3]] = info[0][iter - 1] = iter;
    info[1][iter] = realValue[3];
    info[2][iter] = iter - 1;
    if (info[1][iter] > info[2][iter]) swap(info[1][iter], info[2][iter]);
}

// Reforming the rooted carving decomposition from the info array
Tree_Node<CarvingNodeInfo>** nodes = new Tree_Node<CarvingNodeInfo>** [2*G->nv()];
for (int i = 1; i < 2*G->nv(); ++i) {
    nodes[i] = new Tree_Node<CarvingNodeInfo>();
    nodes[i]->left_child = nodes[i]->right_sibling = NULL;
    if (i < G->nv()) {
        // a leaf
        nodes[i]->data.infoSet.insert(i);
        nodes[i]->data.u = i;
    } else {
        // a node formed through the contraction, it must be an internal one
        nodes[i]->data.u = 0;
    }
}

// fix the tree pointers to childs and siblings using the info array (indices)
for (int i = 1; i < 2*G->nv(); ++i) {
    if (info[1][i] && info[2][i]) {
        nodes[i]->left_child = nodes[info[1][i]];
        nodes[i]->right_sibling = nodes[info[2][i]];
    }
    if (info[0][i] == 0) {
        // found the root
        root = nodes[i];
    }
}

// delete the pointer holder array
delete[] nodes;

// delete the information array for edge contractions
for (int i = 0; i < 3; ++i)
    delete[] info[i];

checkTime("An optimal carving decomposition found in");
// construct the info vertex set (V_i) for each node
formInfoVSet(root_ptr());
checkTime("Constructing info vertex sets in");
// construct the midset for each node
formInfoMidSet(root_ptr());
checkTime("Constructing midsets in");

// clean up the memory
for (int i = 1; i <= G->ne(); ++i) delete[] D[i];
// return TRUE if the carving width of the given graph is LESS THAN k

bool Carving_Decomposition::ST(TopologicalGraph G, int k) {
    ++STCalls;
    // Step 1: check max degree
    int dmin, dmax;
    G.MinMaxDegree(dmin, dmax);
    if (dmax >= k) return false;

    int V, E;
    E = G.ne();
    V = G.nv();
    // Step pre-2 [a]: find the dual
    // Note: edge indices in the dual are exactly the same
    // as in the original graph
    GraphContainer* dGC = G.DualGraph();
    TopologicalGraph dG (*dGC);

    // Step pre-2 [b]: find all pair shortest paths in dG
    for (tvertex i = 1; i <= dG.nv(); ++i)
        for (tvertex j = 1; j <= dG.nv(); ++j)
            D[i()][j()] = dG.nv() + 1;

    delete D;
    delete Q;
    delete cGC;
    return true;
}

// called inside findCarvingDecomposition to form the vertex set under each node
void Carving_Decomposition::formInfoVSet (iterator p) {
    // info set has the set of edges for the mid set of the decomposition edge above current node
    Tree_Node<CarvingNodeInfo>* ptr = p.real_val();
    if (ptr->left_child != NULL) {
        Tree_Node<CarvingNodeInfo>* rightChild = ptr->left_child->right_sibling;
        formInfoVSet (ptr->left_child);
        ptr->data.infoSet.insert(ptr->left_child->data.infoSet.begin(), ptr->left_child->data.infoSet.end());
        ptr->data.infoSet.insert(rightChild->data.infoSet.begin(), rightChild->data.infoSet.end());
    }
}

// has to be called after InfoVSet is called
// midset contains edge indices according to the PIGALE’s Graph class
void Carving_Decomposition::formInfoMidSet (iterator p) {
    set<size_t> res;
    Tree_Node<CarvingNodeInfo>* ptr = p.real_val();
    for (tedge e = 1; e <= G->ne(); ++e) {
        int u = G->vin [−e]();
        int v = G->vin [ e]();
        int cnt = 0;
        if (ptr->data.infoSet.find(u) != ptr->data.infoSet.end()) ++cnt;
        if (ptr->data.infoSet.find(v) != ptr->data.infoSet.end()) ++cnt;
        if (cnt == 1) res.insert (e());
    }
    ptr->data.infoSet = res;
    for (iterator p1 = first_child (p); p1 != end(); ++p1)
        formInfoMidSet (p1);
}

// the critical function: ST-Procedure
// return TRUE if the carving width of the given graph is LESS THAN k

void Carving_Decomposition::formInfoMidSet (iterator p) {
    set<size_t> res;
    Tree_Node<CarvingNodeInfo>* ptr = p.real_val();
    for (tedge e = 1; e <= G->ne(); ++e) {
        int u = G->vin [−e]();
        int v = G->vin [ e]();
        int cnt = 0;
        if (ptr->data.infoSet.find(u) != ptr->data.infoSet.end()) ++cnt;
        if (ptr->data.infoSet.find(v) != ptr->data.infoSet.end()) ++cnt;
        if (cnt == 1) res.insert (e());
    }
    ptr->data.infoSet = res;
    for (iterator p1 = first_child (p); p1 != end(); ++p1)
        formInfoMidSet (p1);
}
// initialize the DP array
for (tvertex i = 1; i <= dG.nv(); ++i) {
    tedge e = dG.FirstEdge(i);
    while (e != 0) {
        tvertex j = dG.win[e];
        if (j == i) j = dG.win[-e];
        D[i][j] = D[j][i] = 1;
        e = dG.NextEdge(i, e);
    }
}
// find all-pair shortest paths by running BFS on each vertex
for (tvertex x = 1; x <= dG.nv(); ++x) {
    Prop<int> mark(dG.Set(tvertex()), PROP_RESERVED);
    memset(Q, 0, sizeof(tvertex));
    mark[x] = 1;
    D[x][x] = 0;
    int qf, ql;
    qf = ql = 0;
    Q[qf] = x();
    while (qf <= ql) {
        tvertex c = Q[qf];
        ++qf;
        mark[c] = 1;
        tbrin first = dG.pbrin[c];
        tbrin b = first;
        do {
            tvertex a = dG.win[-b];
            if (!mark[a]) {
                mark[a] = 1;
                D[a][x] = D[x][a] = min(D[x][a], (int)(D[x][c] + 1));
                ++ql;
                Q[ql] = a();
            }
        } while ((b = dG.cir[b]) != first);
    }
}
// Step pre-2[c]: number faces and link edges to them
// this makes it easier when it comes to the main section of
// the algorithm when we need to know which faces incident to
// an edge and vice versa
Prop<tvertex> faces(G.Set(tbrin()), PROP_RESERVED);
// 2 brin (-e,e) make up an edge => 1 value is for the left face,
// the other is for the right face
void faces.clear() {
    for (tvertex e = -E; e <= E; ++e) {
        if (faces[e]() || !e()) continue;
        tvertex ee = e;
        while (ee != e) {
            faces[ee] = F;
            ee = G.cir[ee];
        }
    }
}
// array used to store information about connected components of G
Prop<vector<int> > nccCount(G.Set(tedge()), PROP_RESERVED + 9);
Prop<vector<int> > nccIdxStart(G.Set(tedge()), PROP_RESERVED + 10);
Prop<vector<int> > nccIdx(G.Set(tedge()), PROP_RESERVED + 11);
nccCount.clear();
nccIdxStart.clear();
nccIdx.clear();

// indices of faces in G is the same for the corresponding vertices in dG
// indices of edges in G is the same for the corresponding edges in dG
// indices of vertices in G MAY NOT be the same for corresponding faces in dG

// Step pre-2 [d]: find Ge for each e
Prop<int> ncc(G.Set(tedge()), PROP_RESERVED + 1); // number of connected components of G for each edge
Prop<vector<int>> mark(G.Set(tedge()), PROP_RESERVED + 2); // mark each vertex belongs to which CC of G for each edge
ncc.clear();
mark.clear();

for (tedge e = 1; e <= E; ++e) {
    Destroy(mark[e]);
    mark[e] = svector<int>(1, V, sizeof(int));
    ncc[e] = GenerateGe(G, e, k, D, mark[e]);
    //mark[e] = tmp;
}

// stack used to store the processing losing states
vector<pair<int, int>> L;

//Pre-2[e]: prepare Xe and Xr
// Xe: set of all game states for edges
// Xr: set of all game states for faces
Prop<int> CountXe(G.Set(tedge()), PROP_RESERVED + 3); // count # of elements for Xe
Prop<vector<int>> CountRE(G.Set(tbrin()), PROP_RESERVED + 4); // count # of elements for (e, r, C), since there are 2 r's for each e, each of them corresponds to 1 brin of edge e (-e and e)
Prop<int> CountXr(dG.Set(tvertex()), PROP_RESERVED + 5); // count # of elements for Xr (or Xv for dG)
Prop<vector<bool>> MarkXe(G.Set(tedge()), PROP_RESERVED + 6); // marker to determine whether (e, C) is still in Xe
Prop<vector<bool>> MarkXr(dG.Set(tvertex()), PROP_RESERVED + 7); // marker to determine whether (r, v) is still in Xr

CountXe.clear();
CountXr.clear();
CountRE.clear();
MarkXe.clear();
MarkXr.clear();

// initialize edge-related arrays

for (tedge e = 1; e <= E; ++e) {
    int C = ncc[e];
    CountXe[e] = C;
    Destroy(CountRE[e]);
    Destroy(CountRE[-e]);
    Destroy(MarkXe[e]);
    Destroy(nccIdxStart[e]);
    Destroy(nccCount[e]);
    MarkXe[e] = svector<bool>({1, C, true});

    if (C < V) {
        CountRE[e] = svector<int>({1, C, (int)0});
        CountRE[-e] = svector<int>({1, C, (int)0});
        nccIdxStart[e] = svector<int>({1, C});
nccCount[e] = svector<int>(1, C);
for (tvertex v = 1; v <= V; ++v)
  ++CountRE[e][mark[e][v]],
  ++CountRE[-e][mark[e][v]]
;

for (int i = 1; i <= C; ++i) {
  nccCount[e][i] = CountRE[e][i];
  nccIdxStart[e][i] = ((i == 1) ? (nccIdxStart[e][i - 1] + nccCount[e][i - 1]) : 0);
} else {
  for (tvertex v = 1; v <= V; ++v)
    mark[e][v] = v();
}

// initialize face-related arrays
for (tvertex r = 1; r <= F; ++r)
  CountXr[r] = V;
  MarkXr[r] = svector<bool>(1, V, true);

// step 3, modify Xr (marking the base case losing states)
for (tbrin b = -E; b <= E; ++b)
  if (!b()) continue;
  tvertex r = faces[b];
  tvertex u, v;
  u = G.vin[-b];
  v = G.vin[b];
  // put (r,u) & (r,v) to stack L and delete them from Xr
  if (MarkXr[r][u]) {
    MarkXr[r][u] = false;
    --CountXr[r];
    L.push_back(make_pair(E + r(), u()));
  }
  if (MarkXr[r][v]) {
    MarkXr[r][v] = false;
    --CountXr[r];
    L.push_back(make_pair(E + r(), v()));
  }
  if (CountXr[r] == 0) {
    L.clear();
    clearCustomizedGraph(G, dG, dGC);
    return true;
  }

int saved;

// step 3-post, A1 improvement from alx08_015bianz
saved = 0;
for (tvertex r = 1; r <= dG.nv(); ++r)
  // for each region r in G, i.e. vertex r in dG
  // go through each vertex in G
for (tvertex v = 1; v <= G.nv(); ++v)
  if (MarkXr[r][v]) {
    // for each vertex in G, go through the edges/faces surrounding it
    tbrin first = G.pbrin[v];
    tbrin b = first;
    vector<int> spath;
    do {

tvertex x = faces [b];
spath.push_back (D [r()] [x()]);
} while ((b = G.cir [b]) != first);
int satisfied = 0;
int spathLen = spath.size ();
for (int i = (spathLen + 1) / 2; l < spathLen && !satisfied; ++l)
for (int i = 0; i < spathLen && !satisfied; ++i)
if (spath [i] + spath [(i + 1) % spathLen] + l < k)
satisfied = 1;
// check for the condition
if (satisfied) {
++saved;
MarkXr [r][v] = false;
--CountXr [r];
L.push_back (make_pair (E + r(), v()));
if (CountXr [r] == 0) {
L.clear ();
clearCustomizedGraph (G, dG, dGC);
return true;
}
}

// step 4
// spreading the losing status over the bipartite graph
while (!L.empty ()) {
pair<int, int> x = L.back ();
L.pop_back ();
if (x.first > E) {
// when x is (r, v)
// loop through edges of r, which are edges circulating around the
corresponding vertex in dG
tvertex r = (x.first - E); // r is a face in G, also a vertex in dG
tvertex v = x.second; // v is a vertex of G
tbrii first = dG.pbrin [r];
tbrii b = first;
do {
edge e = (edge) abs (b());
// if there is a state (e,C) in Xe with v in C
int C = mark [e][v];
if (MarkXe [e][C]) {
// if there may be connected components with > 1 vertex
// then we use the countRE property
// by decreasing c(r,e,C)
--CountRE [b][C];
}
// if c(r,e,C) becomes 0, move (e,C) from Xe to L
if (ncc [e] == V || CountRE [b][C] == 0) {
L.push_back (make_pair (e(), C));
MarkXe [e][C] = false;
--CountXe [e];
if (CountXe [e] == 0) {
L.clear ();
clearCustomizedGraph (G, dG, dGC);
return true;
}
}
}
while ((b = dG.cir [b]) != first);
} else {
    // when x is (e, C)
    tedge e = (x.first);
    int C = x.second;
    // check each of the 2 faces incident to e
    for (int i = -1; i <= 1; i+=2) {
        tbin b = e () * i;
        tvertex r = faces [b];
        // special case, when each vertex is in a separate connected component
        // then we don't need CountRE, nccCount, nccIdx, and nccIdxStart
        if (ncc [e] == r) {
            tvertex v = (tvertex) C;
            if (MarkXr [r][v]) {
                L.push_back (make_pair (E + r(), v()));
                MarkXr [r][v] = false;
                −CountXr [r];
                if (CountXr [r] == 0) {
                    L.clear ();
                    clearCustomizedGraph (G, dG, dGC);
                    return true;
                }
            }
        }
    }
}
else {
    // if c(r, e, C) > 0
    if (CountRE[b][C]) {
        // for each vertex v of C s.t. (r, v) in Xr
        for (int i = 0; i < nccCount [e][C]; ++i) {
            tvertex v = nccIdx[e][nccIdxStart[e][C] + i];
            // move (r, v) from Xr to L
            L.push_back (make_pair (E + r(), v()));
            MarkXr [r][v] = false;
            −CountXr [r];
            if (CountXr [r] == 0) {
                L.clear ();
                clearCustomizedGraph (G, dG, dGC);
                return true;
            }
        }
    }
}

// if any of X_e or X_r is empty (i.e., the rat–catcher wins)
// it should have been caught by the pre steps or during the while
// loop and the YES answer is returned immediately

// when the flow goes to here, that means all of the set X_e
// and X_r are not empty, therefore the rat–catcher cannot win
// the game, simply do a clean up and return FALSE
L.clear ();
clearCustomizedGraph (G, dG, dGC);
return false;

// construct the graph Ge & its connected components
// vertices of each component are marked by the index in mark
// return # of connected components
```c
int Carving_Decomposition::GenerateGe (TopologicalGraph& G, tedge e, int k, int** D, svector<int>& mark) {
    int res;
    // instead of constructing individual graph G_e, we just run
    // BFS on G itself while ignoring all the noisy edges with the
    // information from the all-pair-shortest-paths array from D
    Prop<tvertex> faces(G.PB(), PROP_RESERVED);
    tvertex u, v, x, y;
    u = faces [e];
    v = faces [-e];
    Prop<svector<int>> nccIdx(G.Set(tedge()), PROP_RESERVED + 11);
    Destroy (nccIdx [e]);
    nccIdx [e] = svector<int> (1,G.nv(),(int)0);
    mark.clear();
    for (tvertex s = 1; s <= G.nv(); ++s) mark [s] = 0;
    int qf, ql;
    qf = ql = 1;
    res = 0;
    // BFS from each unmarked vertex while ignoring noisy edges
    for (tvertex s = 1; s <= G.nv(); ++s) if (mark [s] == 0) {
        ++res;
        mark [s] = res;
        ql = qf;
        nccIdx [e][qf] = s();
        mark [s] = res;
        while (qf <= ql) {
            tvertex now = nccIdx [e][qf];
            ++qf;
            tbrin first = G.pbrin [now];
            tbrin b = first;
            do {
                tvertex other = G.vin[-b];
                x = faces [b];
                y = faces [-b];
                if (D[u()][x()][y()][z()] + D[u()][y()][z()] + 2 >= k &
                    & D[u()][y()][z()] + D[x()][y()][z()] + 2 >= k &
                    & mark [other] == 0) {
                    mark [other] = res;
                    ++ql;
                    nccIdx [e][ql] = other();
                }
                while ((b = G.cir [b]) != first);
            }
        }
    }
    if (res == G.nv()) Destroy (nccIdx [e]);
    //nccIdx[e] = Q;
    return res;
}
```

// clean up any dynamic properties we created before

```c
void Carving_Decomposition::clearCustomizedGraph (TopologicalGraph& G, TopologicalGraph& dG, GraphContainer* dGC) {
    Prop<int> ncc(G.Set(tedge())), PROP_RESERVED + 1); // number of connected components
    // of G_e for each edge
    for (tedge e = 1; e <= G.ne(); ++e) Destroy (mark [e]);
```

```c
```
Prop<vector<bool>> MarkXe(G.Set(tedge())), PROP_RESERVED + 6); // marker to determine whether (e,C) is still in Xe
for (tedge e = 1; e <= G.ne(); ++e) Destroy (MarkXe [e]);
Prop<vector<bool>> MarkXr(dG.Set(tvertex())), PROP_RESERVED + 7); // marker to determine whether (r,v) is still in Xr
for (tvertex v = 1; v <= G.nv(); ++v) Destroy (MarkXr [v]);
Prop<vector<int>> CountRE(G.Set(tbrin())), PROP_RESERVED + 4); // count # of elements for (e,r,C), since there are 2 r’s for each e, each of them corresponds to 1 brin of edge e (−e and e)
Prop<vector<int>> nccCount(G.Set(tedge())), PROP_RESERVED + 9)
Prop<vector<int>> nccIdxStart(G.Set(tedge())), PROP_RESERVED + 10)
Prop<vector<int>> nccIdx(G.Set(tedge())), PROP_RESERVED + 11)
for (tedge e = 1; e <= G.ne(); ++e)
if (ncc [e] < G.nv()) {
    Destroy (CountRE [−e]); Destroy (CountRE [e]);
    Destroy (nccIdxStart [e]);
    Destroy (nccCount [e]);
    Destroy (nccIdx [e]);
}
// erase all properties
G.Set(tbrin()).erase(PROP_RESERVED + 0);
G.Set(tedge()).erase(PROP_RESERVED + 1);
G.Set(tedge()).erase(PROP_RESERVED + 2);
G.Set(tedge()).erase(PROP_RESERVED + 3);
G.Set(tbrin()).erase(PROP_RESERVED + 4);
G.Set(tedge()).erase(PROP_RESERVED + 6);
G.Set(tedge()).erase(PROP_RESERVED + 9);
G.Set(tedge()).erase(PROP_RESERVED + 10);
G.Set(tedge()).erase(PROP_RESERVED + 11);
dG.Set(tvertex()).erase(PROP_RESERVED + 0);
dG.Set(tvertex()).erase(PROP_RESERVED + 5);
dG.Set(tvertex()).erase(PROP_RESERVED + 7);
dG.reset();
dGC->clear();
delete dGC;
}

A.2 Class Branch_Deployment

Listing A.3: Branch_Deployment.h
```cpp
#include "Tree.h"
#include "BaseGraph.h"
#include "Carving_Decomposition.h"
#include "Tree_Decomposition.h"

#include <Pigale.h>
#include <TAXI/netcut.h>
#include <map>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cassert>
#include <vector>
#include <set>
#include <stack>
#include <list>
#include <ctime>

using namespace std;

struct BranchNodeInfo {
    size_t u, v;
    set<size_t> infoSet;
};

class Branch_Decomposition: public Tree<BranchNodeInfo> {

public:
    bool build_decomposition(BaseGraph &G);
    bool build_decomposition(TopologicalGraph *TG);
    Tree_Decomposition* toTreeDecomposition();

    int width() {
        return rec_width(root_ptr());
    }

    void print() {
        int counter = -1;
        rec_print(root_ptr(), counter);
    }

    Branch_Decomposition::Branch_Decomposition() {
        is_formed = false;
        G = NULL;
    }

protected:
    Tree_Node<BranchNodeInfo>* rec_FromCarvingDecomposition (Tree_Node<CarvingNodeInfo> * p);
    void rec_ToTreeDecomposition(Tree_Decomposition* D, Tree_Decomposition::iterator parent, iterator p);

    // helper functions
    void formInfoESet (iterator p);
    void formInfoMidSet (iterator p);
    void fixIndex (iterator p);

private:
    TopologicalGraph* G;
    bool is_formed;
    bool firstCalled;
    clock_t lastTime;
    void checkTime(string status = "Check time");
    void rec_print(iterator p, int &counter) {
```
iterator p2;
list<int> sub_tree_nums;
list<int>::iterator p1;
set<size_t>::iterator p3;

if (p==end()) {
    return;
}
for (p2 = first_child(p); p2!=end(); ++p2) {
    rec_print(p2, counter);
    sub_tree_nums.push_back(counter);
    counter++;
}

if (sub_tree_nums.empty()) {
    cout << counter << "uL_u" << "(p).u" << "(p).v" << "u(";
    for (p3=(p).infoSet.begin(); p3!=(p).infoSet.end(); ++p3) {
        cout << "u" << "*p3;"
    }
    cout << "u)" << endl;
} else {
    cout << counter << "uI_u{";
    for (p3=(p).infoSet.begin(); p3!=(p).infoSet.end(); ++p3) {
        cout << "u" << "*p3;
    }
    cout << "u}u{";
    for (p1=sub_tree_nums.begin(); p1!=sub_tree_nums.end(); ++p1) {
        cout << "u" << "*p1;
    }
    cout << "u)}" << endl;
}

int rec_width(Branch_Decomposition::iterator p) {
    int max, temp;
    Branch_Decomposition::iterator p1;
    if (p!=end()) {
        max = (*p).infoSet.size();
        for (p1=first_child(p); p1!=end(); ++p1) {
            temp = rec_width(p1);
            if (max < temp) max = temp;
        }
        return max;
    }
    return 0;
}
#include <iostream>

using namespace std;

bool Branch_Decomposition::build_decomposition(BaseGraph &BG) {
    // first, build the Graph object (from PIGALE library)
    // and then call the other function on this object
    return build_decomposition(BG.get_TopologicalGraph());
}

bool Branch_Decomposition::build_decomposition(TopologicalGraph *TG) {
    firstCalled = true;
    G = TG;
    if (!G->FindPlanarMap()) return false;
    // at this point we know that G is planar

    // build the medial graph
    GraphContainer* mGC = MedialGraph(*G);
    TopologicalGraph mG (*mGC);
    // use the carving to build branch
    Carving_Decomposition carving;

    checkTime();
    // construct the carving decomposition of the medial graph
    carving.build_decomposition(&mG);

    // Carving_Decomposition found, convert it to Branch_Decomposition
    root = rec_FromCarvingDecomposition (carving.root_ptr().real_val());

    // construct the info set for each node
    formInfoESet (root_ptr());

    // construct the midset for each node
    formInfoMidSet (root_ptr());

    // fix the indices of the edges
    fixIndex (root_ptr());
    checkTime("Finished constructing an optimal branch decomposition in");
    is_formed = true;
    return true;
}

void Branch_Decomposition::checkTime(string status) {
    clock_t currentTime = clock();
    if (!firstCalled) {
        cerr << status << ": " << ((double)(currentTime - lastTime))/CLOCKS_PER_SEC << " elapsed;";
    } else {
        firstCalled = false;
    }
    lastTime = currentTime;
}

// recursively convert a subtree of a carving decomposition to a branch decomposition
Tree_Node<BranchNodeInfo>* Branch_Decomposition::rec_FromCarvingDecomposition (Tree_Node<BranchNodeInfo>* p) {
    if (p == NULL) return NULL;
    Tree_Node<BranchNodeInfo>* res = new Tree_Node<BranchNodeInfo>();
    res->data.u = p->data.u;
    res->data.v = 0;
    res->left_child = res->right_sibling = NULL;
    res->data.infoSet.clear();
    return res;
}
if (p->left_child != NULL) res->left_child = rec_FromCarvingDecomposition (p->left_child);
else res->data.infoSet.insert(res->data.u);
if (p->right_sibling != NULL) res->right_sibling = rec_FromCarvingDecomposition (p->right_sibling);
return res;
}

// called inside buildDecomposition to form the edge indices set below each node
void Branch_Decomposition::formInfoESet (iterator p) {
    Tree_Node<BranchNodeInfo>* ptr = p.p.real_val();
    if (ptr->left_child != NULL) {
        Tree_Node<BranchNodeInfo>* rightChild = ptr->left_child->right_sibling;
        formInfoESet (ptr->left_child);
        formInfoESet (rightChild);
        ptr->data.infoSet.clear ();
        ptr->data.infoSet.insert(ptr->left_child->data.infoSet.begin(), ptr->left_child->data.infoSet.end());
        ptr->data.infoSet.insert(rightChild->data.infoSet.begin(), rightChild->data.infoSet.end());
    }
}

// has to be called after InfoESet is called
// midset contains vertex indices according to the PIGALE's Graph class
// info set has the set of vertices for the mid set of the decomposition edge above current node
void Branch_Decomposition::formInfoMidSet (iterator p) {
    set<size_t> res;
    Tree_Node<BranchNodeInfo>* ptr = p.p.real_val();
    for (tree<vertex> v = 1; v <= G->nv (); ++v) {
        pbrin first = G->pbrin [v];
        pbrin b = first;
        int cnt = 0;
        int matched = 0;
        do {
            if (ptr->data.infoSet.find(abs(b())) != ptr->data.infoSet.end()) ++matched;
            ++cnt;
        } while ((b = G->cir [b]) != first);
        if (matched > 0 & & matched < cnt) res.insert(v());
    }
    ptr->data.infoSet = res;
    for (iterator p1 = first_child (p); p1 != end(); ++p1)
        formInfoMidSet (p1);
}

// because PIGALE use 1-based indices while our interface use 0-based indexes
// we need to fix the index at the end of the construction
void Branch_Decomposition::fixIndex (iterator p) {
    Tree_Node<BranchNodeInfo>* ptr = p.p.real_val();
    for (std::set<vertex> ::iterator itr = ptr->data.infoSet.begin ();
        itr != ptr->data.infoSet.end (); ++itr) {
        newBag.insert(*itr - 1);
    }
    ptr->data.infoSet = newBag;
    for (ptr->left_child != NULL) {
        // this is an internal node, u & v doesn't matter
        ptr->data.u = ptr->data.v = G->nv();
    }
for (iterator p1 = first_child (p); p1 != end(); ++p1)
    fixIndex (p1);
} else {
    tbrin b = ptr->data.u;
    ptr->data.v = G->vin [b] () - 1;
    ptr->data.u = G->vin [~b] () - 1;
}
}

// convert a branch decomposition to a tree decomposition
Tree_Decomposition* Branch_Decomposition::toTreeDecomposition () {
    if (!is_formed) return NULL;
    Tree_Decomposition* res = new Tree_Decomposition ();
    rec_ToTreeDecomposition (res, Tree_Decomposition::iterator(NULL), root_ptr());
    return res;
}

// recursively convert a subtree in a branch decomposition to a tree decomposition
void Branch_Decomposition::rec_ToTreeDecomposition (Tree_Decomposition* D, Tree_Decomposition::iterator parent, iterator p) {
    Tree_Node<BranchNodeInfo>* ptr = p.real_val();
    set<size_t> bag;
    if (ptr->left_child == NULL) {
        // leaf node
        bag.insert(ptr->data.u);
        bag.insert(ptr->data.v);
    } else {
        // internal node
        Tree_Node<BranchNodeInfo>* leftChild = ptr->left_child;
        Tree_Node<BranchNodeInfo>* rightChild = leftChild->right_sibling;
        set<size_t> CHILDREN;  // union set of: left child and right child
        set_union (leftChild->data.infoSet.begin (), leftChild->data.infoSet.end (),
                   rightChild->data.infoSet.begin (), rightChild->data.infoSet.end (),
                   insert_iterator<set<size_t>> (CHILDREN, CHILDREN.begin ()));
        set_union (CHILDREN.begin (), CHILDREN.end (),
                   ptr->data.infoSet.begin (), ptr->data.infoSet.end (),
                   insert_iterator<set<size_t>> (bag, bag.begin ()));
    }
    // insert the node into the given tree
    Tree_Decomposition::iterator currentNode;
    if (parent.real_val() != NULL)
        currentNode = D->insert_child_back (bag, parent);
    else
        currentNode = D->add_root (bag);
    for (iterator p1 = first_child (p); p1 != end(); ++p1)
        rec_ToTreeDecomposition (D, currentNode, p1);
}

A.3 Helper Functions

Listing A.5: GraphUtils.h
FILE: GraphUtils.h
AUTHOR: Hiêp Dinh
DATE: July 2008

DESCRIPTION: GraphUtils contains a set of functions related to the Graph class from PIGALE library

#ifndef GRAPH_UTILS
#define GRAPH_UTILS
#include <Pigale.h>
#include <TAXI/netcut.h>
#include <iostream>

using namespace std;

GraphContainer* MedialGraph (TopologicalGraph &G);
void printGraph (const TopologicalGraph &G, ostream& outs = cout);

// fix compatibility issue for the svector<T> class, don’t use this function
template<class T>
ostream& operator<<(ostream& outs, const svector<T> v) {
    for (int i = 0; i < v.n(); ++i)
        outs << (i ? "\n" : " ") << v[i];
    return outs;
}

// fix compatibility issue, don’t use this function
template<class T>
istream& operator>>(istream& ins, const svector<T> v) {
    return ins;
}
#endif

Listing A.6: GraphUtils.cc

FILE: GraphUtils.h
AUTHOR: Hiêp Dinh
DATE: July 2008

DESCRIPTION: GraphUtils contains a set of functions related to the Graph class from PIGALE library

#include "GraphUtils.h"

GraphContainer* MedialGraph (TopologicalGraph &G) {
    if(debug())DebugPrintf("Executing\nMedialGraph");
    if(!G.CheckConnected() || !G.FindPlanarMap()) {
        DebugPrintf("Could\nnot\ncompute\nmedial\ngraph");
        setPigaleError(-1,"Could\nnot\ncompute\nmedial\ngraph");
        return (GraphContainer *)0;
    }

    int m = G.ne();
    int n = G.nv();
    int nn = m; // each original edge corresponds to a new vertex
    int nm = 0; // each vertex v with degree > 1 corresponds to deg(v) edges
    for (int i = 1; i <= n; ++i) {
        int degi = G.Degree (i);
        if (degi > 1) nm += degi;
if (debug()) DebugPrintf("MedialGraph:n:%d,m:%d",nn,mm);
GraphContainer & Medial = *new GraphContainer;
Medial.setSize(nn,mm);
Prop1<cstring> title(G.Set(),PROP_TITRE);
Prop1<cstring> titleD(Medial.Set(),PROP_TITRE);
titleD() = "M-" + title();

Prop1<tvertex> mvin(Medial.PB(),PROP_VIN);
//mvin.clear();
mvin.SetName("medial:mvin");

tvertex v;
tbrin mbrin = 0;
for (v = 1; v <= n; ++v) {
    tbrin first = G.pbrin[v];
    tbrin b = first;
    tbrin lastb;
    do {
        lastb = b;
        b = G.cir[b];
        if (lastb == b) break;
        ++mbrin;
        mvin[+mbrin] = (tvertex) abs(lastb());
        mvin[-mbrin] = (tvertex) abs(b());
    } while (b != first);
}
if (mbrin != nm) {
    DebugPrintf("m=%d,n=%d",nm,nn);
    setPigaleError(A_ERRORS_DUAL); // should have a separate error code for MEDIAL
    DebugPrintf("Error Computing the medial: mbrin != fixed-edges %d!=%d",mbrin(),nm);
    delete &Medial;
    return (GraphContainer *)0;
}
if (debug()) DebugPrintf("END_MedialGraph");
return &Medial;

void printGraph (const TopologicalGraph &G, ostream& outs) {
    // print the number of vertices and edges
    outs << "Nodes:" << G.nv() << "Edges:" << G.ne() << endl;
    // print the edges (if e is a tedge, e() is the int that represents it)
    outs << "Edges:" << endl;
    for (tedge e = 1; e <= G.ne(); e++)
        outs << e() << "[" << G.vin[e]() << "," << G.vin[-e]() << "]" << endl;
    // At each vertex v there is a tbrin G.pbrin[v] incident to it: G.vin[G.pbrin[v]] = v;
    // So we can print the planar map (circular order of half edges around each vertex)
    outs << "Map_\(half\_edges\):" << endl;
    for (tvertex v = 1; v <= G.nv(); v++)
    {
        outs << v() << "->";
        tbrin first = G.pbrin[v];
        tbrin b = first;
        do
        {
            outs << b() << ";";
        } while((b = G.cir[b]) != first);
        outs << endl;
    } // Print the circular order of vertices around each vertex
    outs << "Map_\(vertices\):" << endl;
for(tvertex v = 1; v <= G.nv(); v++)
    { outs << v() << "u";
        tbrin b = G.pbrin[v];
        outs << b;
        do
            { outs << G.vin[-b]();
                while((b = G.cir[b]) != first);
            }
        outs << endl;
    }

    // Print the circular order of vertices around each vertex
    outs << "Adj(vertcises):" << endl;
    for(tvertex v = 1; v <= G.nv(); v++)
        { outs << v() << ":u";
            tedge e = G.FirstEdge(v);
            while (e != 0) 
                { outs << ["v" << G.vin[-e]()] << "," << G.vin[e]() ;
                    e = G.NextEdge(v,e);
                }
            outs << endl;
        }