1990

Efficient processing of hierarchical graphs

Mark Allen Williams
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
Williams, Mark Allen, "Efficient processing of hierarchical graphs " (1990). Retrospective Theses and Dissertations. 9385.
http://lib.dr.iastate.edu/rtd/9385

This Dissertation is brought to you for free and open access by Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Efficient processing of hierarchical graphs

Williams, Mark Allen, Ph.D.

Iowa State University, 1990
Efficient processing of hierarchical graphs

by

Mark Allen Williams

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:
Signature was redacted for privacy.

In Charge of Major Work
Signature was redacted for privacy.

For the Major Department
Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1990
# TABLE OF CONTENTS

## ACKNOWLEDGEMENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Related Work</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 The hierarchical input language</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 The small circuit model</td>
<td>4</td>
</tr>
<tr>
<td>1.1.3 The dynamic graph model</td>
<td>5</td>
</tr>
<tr>
<td>1.2 What Follows</td>
<td>6</td>
</tr>
</tbody>
</table>

## 2. PRELIMINARIES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Graph Theory</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1 Basic terminology and concepts</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2 Subgraphs, minors, and homeomorphism</td>
<td>11</td>
</tr>
<tr>
<td>2.1.3 Graph connectivity</td>
<td>12</td>
</tr>
<tr>
<td>2.2 The Hierarchical Graph Model</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Hierarchical graphs</td>
<td>18</td>
</tr>
<tr>
<td>2.2.2 Comparing the sizes of $\Gamma$ and $X(\Gamma)$</td>
<td>22</td>
</tr>
<tr>
<td>2.2.3 Previous results on hierarchical graphs</td>
<td>24</td>
</tr>
</tbody>
</table>

## 3. PROCESSING HIERARCHICAL GRAPHS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 The Bottom-Up Method</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1 Algorithm BU</td>
<td>27</td>
</tr>
</tbody>
</table>
### 5. SUBGRAPH HOMEOMORPHISM PROBLEMS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Replaceability and Edge Deletion and Contraction</td>
<td>77</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Contraction in series-parallel graphs</td>
<td>78</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Contraction in outer-planar graphs</td>
<td>83</td>
</tr>
<tr>
<td>5.2</td>
<td>Recognition of Series-Parallel Graphs</td>
<td>86</td>
</tr>
<tr>
<td>5.2.1</td>
<td>The burner</td>
<td>86</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Run-time analysis</td>
<td>93</td>
</tr>
<tr>
<td>5.3</td>
<td>Recognition of Outer-Planar Graphs</td>
<td>95</td>
</tr>
<tr>
<td>5.3.1</td>
<td>The burner</td>
<td>95</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Run-time analysis</td>
<td>97</td>
</tr>
<tr>
<td>5.4</td>
<td>Extracting a Forbidden Subgraph</td>
<td>99</td>
</tr>
<tr>
<td>5.4.1</td>
<td>The basic idea</td>
<td>100</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Algorithm GEN</td>
<td>103</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Implementation details</td>
<td>107</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>108</td>
</tr>
</tbody>
</table>

### 6. MATROID OPTIMIZATION PROBLEMS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Matroids and The Greedy Algorithm</td>
<td>110</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Definitions</td>
<td>110</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Properties of matroids</td>
<td>112</td>
</tr>
<tr>
<td>6.1.3</td>
<td>A variant of the greedy algorithm</td>
<td>113</td>
</tr>
<tr>
<td>6.2</td>
<td>A Greedy Algorithm for Hierarchical Graphs</td>
<td>116</td>
</tr>
<tr>
<td>6.2.1</td>
<td>The burner</td>
<td>116</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Algorithm H-Greedy</td>
<td>117</td>
</tr>
<tr>
<td>6.3</td>
<td>Matroidal Families</td>
<td>120</td>
</tr>
<tr>
<td>6.4</td>
<td>Cost-Finding Algorithms</td>
<td>123</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

| Figure 1.1: | An HIL description and the rectangles it generates | 4 |
| Figure 1.2: | Static graph $G$ and induced dynamic graph $G^2$ | 7 |
| Figure 2.1: | A graph and some of its minors | 12 |
| Figure 2.2: | Graph $G$ and its bc-forest $bcf(G)$ | 13 |
| Figure 2.3: | Graph $G$ and its br-forest $brf(G)$ | 14 |
| Figure 2.4: | Directed graph $G$ and its st-dag $std(G)$ | 15 |
| Figure 2.5: | (a) $G$, (b) the triconnected components of $G$, and (c) $tct(G)$ | 17 |
| Figure 2.6: | $\Gamma = (G_1, G_2, G_3)$ and its expansions | 21 |
| Figure 2.7: | The hierarchy tree of $\Gamma$ from Figure 2.6 | 22 |
| Figure 3.1: | Algorithm BU | 28 |
| Figure 3.2: | The bipartiteness testing burner | 29 |
| Figure 3.3: | The connectivity augmentation burner | 34 |
| Figure 3.4: | Algorithm ConAug | 35 |
| Figure 3.5: | Algorithm GBU | 40 |
| Figure 4.1: | How to glue br-forests | 51 |
| Figure 4.2: | Gluing br-forests via $L = [(a,1), (b,2), (c,3)]$ | 52 |
| Figure 4.3: | The Glue Forests operation | 62 |
| Figure 4.4: | The Condense Blocks operation | 63 |
Figure 4.5: Gluing bc-forests via $L = [(a,1),(b,2),(c,3)]$ .......................... 64

Figure 5.1: The Kuratowski graphs ........................................... 76

Figure 5.2: The graphs $K_4$ and $K_{2,3}$ ....................................... 77

Figure 5.3: The series-parallel burner ........................................ 90

Figure 5.4: Contracting $e$ into $x$ ............................................. 100

Figure 5.5: Expanding vertices .................................................. 102

Figure 5.6: Algorithm GEN ....................................................... 105

Figure 5.7: Procedure FIND-EDGES .......................................... 106

Figure 6.1: Algorithm L-Greedy ............................................... 114

Figure 6.2: The H-Greedy burner .............................................. 118

Figure 6.3: The bicycle graphs ................................................ 120

Figure 6.4: The Base-Cost burner ............................................. 124

Figure 6.5: Algorithm Gen-Base .............................................. 132
ACKNOWLEDGEMENTS

I would like to acknowledge the contributions to this work made by my advisor, Dr. David Fernández-Baca. His knowledge and guidance proved to be invaluable throughout all phases of my research.

I would especially like to thank my parents, Dr. and Mrs. Stanley A. Williams, for their support (and tolerance) during these past four years.
1. INTRODUCTION

The standard encoding of a graph usually consists of a list of its vertices and edges. However, graphs encountered in some domains exhibit structural regularities that allow for more succinct encodings. For example, logic circuits are often constructed by replicating other circuits, a process that results in a natural hierarchical representation of the circuit. Thus, a circuit can be represented by a list of its parts and a set of instructions for assembling the circuit from the parts. This succinct encoding may be substantially smaller than the circuit it describes.

Given the potential difference in size between the standard and succinct representations of a graph, a logical question is whether or not there are graph problems that can be efficiently solved when inputs are succinctly described. The answer is often negative in that problems which are polynomially-solvable under a standard encoding scheme often become NP-hard or worse when inputs are succinctly encoded. An exception is the Hierarchical Graph Model of Lengauer [19]. A hierarchical graph \( \Gamma \) consists of a list of special graphs called cells. The graph represented by \( \Gamma \) is called its \textit{expansion}, and is denoted by \( X(\Gamma) \). \( X(\Gamma) \) is constructed by attaching cells together by a process called \textit{gluing}. Which cells are to be glued together and how the gluing is to be carried out is completely specified within \( \Gamma \). A \textit{hierarchical algorithm} is an algorithm that, given \( \Gamma \), solves some problem defined on \( X(\Gamma) \). The \textit{hierarchical time (space) complexity} of a problem is its time (space) complexity
when inputs are hierarchical graphs. Many efficient hierarchical algorithms have been based on the bottom-up method, a framework developed by Lengauer and Wanke [24] for constructing and analyzing hierarchical algorithms for certain decision problems. A formal description of the bottom-up method appears in Section 2.2.

This work is mainly an attempt to understand the kinds of problems that have efficient hierarchical algorithms. Our effort stems from the observation that only a few of the properties of the bottom-up method and graph gluing are crucial to the correctness of hierarchical algorithms. We generalize those properties so that they may be applied to a wider range of graph problems. This provides us with a uniform approach to constructing hierarchical algorithms and proving their correctness. We develop polynomial time hierarchical algorithms for several augmentation problems [8,31], and the hierarchical recognition of series-parallel [7] and outer-planar [33] graphs. Techniques applied to the latter two problems also allow for the generation of a forbidden subgraph of the expansion of $\Gamma$, when one exists, using work-space linear in the size of $\Gamma$. We characterize two infinite families of graph problems for which polynomial-time hierarchical algorithms exist. Each problem amounts to computing the cost of an optimum base of a matroid defined on the edges of a graph. The classes of matroids we consider were studied by Matthews [26,27], White and Whitely [39], and Simões Pereira [32]. In addition, we develop polynomial-space hierarchical algorithms that generate optimum bases for these matroid classes.

1.1 Related Work

Several other models for succinct representation have been developed in the past. Studies have concentrated on how problem complexity is affected when instances are succinctly encoded. In this section, we briefly discuss three models, each of which
is capable of representing an object using space polylogarithmic in the size of the object.

1.1.1 The hierarchical input language

In general, algorithms that operate on sets of rectangles have as their input a simple list of rectangles. However, according to Bentley, Ottmann and Widmayer [4], such an encoding scheme is unreasonable in problem domains, such as VLSI, where objects have regular structures and are often defined hierarchically.

Bentley et al. [4] provide a language for hierarchically describing sets of rectangles whose sides are parallel to the coordinate axes. They consider a number of polynomially-solvable problems on rectangle sets that are encountered in various stages of the VLSI design and validation process. Their description method is called the Hierarchical Input Language (HIL). An HIL description consists of a sequence of symbols numbered consecutively from zero. Each symbol $i$ describes a set of rectangles using BOX and DRAW commands. The command BOX $(x, y)(w, h)$ draws a rectangle having width $w$, height $h$, and lower left corner at $(x, y)$. The command DRAW $j$ AT $(x, y)$ calls upon the symbol $j < i$ to draw the rectangles it describes relative to the point $(x, y)$. The rectangles described by an HIL description are those generated by its highest numbered symbol. Figure 1.1 shows an HIL description and the rectangles it generates.

With this unrestricted form of HIL, most non-trivial problems are intractable [4]. However, by restricting its expressive power somewhat, several important problems can be efficiently solved. The restricted language, called consistent HIL, is the same as HIL except for the addition of a minimum bounding rectangle (MBR) command within each symbol. The MBR command defines the smallest rectangle enclosing
Figure 1.1: An HIL description and the rectangles it generates

the rectangles described by the symbol. An HIL description is consistent if for each symbol, there are no intersections among the boxes it draws and the minimum bounding rectangles of the symbols it invokes.

Unfortunately, the conciseness of the description is sometimes lost in consistent HIL [4]. Bentley et al. [4] also point out that consistent HIL is too restrictive in the sense that no “real” designs can be described with it.

1.1.2 The small circuit model

Galperin and Widgerson [13] consider succinctly describing a graph by encoding its adjacency structure in a combinatorial circuit that is described as follows:

Let $G$ be a graph with $m \leq 2^n$ vertices $0, \ldots, m - 1$. A combinatorial circuit $C_G$ is called a small circuit representation of $G$ (an SCR of $G$)
provided it has two n-bit inputs, at most \( p(n) \) gates for some polynomial \( p \), and output given by:

\[
C_G(i,j) = \begin{cases} 
? & \text{if } i \text{ or } j \text{ is not a vertex of } G \\
0 & \text{if } (i,j) \text{ is not an edge of } G \\
1 & \text{if } (i,j) \text{ is an edge of } G 
\end{cases}
\]

Galperin and Widgerson prove that almost every nontrivial graph problem is NP-complete when the graph is described by an SCR. For example, it is NP-complete to determine if the graph described by an SCR has an edge. Galperin and Widgerson’s results were strengthened by Papadimitriou and Yannakakis [29], who showed that the complexity of many graph problems increases exponentially under the SCR model.

The small circuit model is different from the other models considered here because it accepts the elements of a graph as opposed to generating the graph [37]. Examining the structure of the circuit provides little or no knowledge of its graph. Instead, one must query the circuit.

1.1.3 The dynamic graph model

A dynamic graph [28] is an infinite graph induced by a finite static graph as follows. Let \( G = (V,E) \) be a directed graph such that each edge \((u,v)\) has an associated integral transit time \( t_{u,v} \). The static graph \( G \) induces the dynamic graph \( G' = (V',E') \) where:

1. \( V' = \{v^p : v \in V \text{ and } p \text{ is an integer}\} \)

2. \( E' = \{(u^p, v^{p+t_{u,v}}) : (u,v) \in E \text{ and } p \text{ is an integer}\} \).
Iwano and Steiglitz [16] consider a $k$-dimensional generalization of these 1-dimensional dynamic graphs that can be used to model complex interconnections of processing elements and several problems occurring in VLSI design checking. A $k$-dimensional dynamic graph $G^k$ is induced by a finite static graph $G = (V, E)$ whose edges are each labeled with a vector of $k$ integers. $G^k$ is obtained by repeating the vertices of $G$ in a $k$-dimensional orthogonal grid and adding edges in a fashion similar to that for 1-dimensional dynamic graphs.

Dynamic graphs can be used to model various scheduling problems. Orlin [28] shows that several nontrivial graph problems can be solved on dynamic graphs by considering only the associated static graph. However, Orlin also provides examples of problems that are easy to solve on finite graphs, but are intractable for dynamic graphs. Figure 1.2 shows a static graph $G$ with 2-dimensional edge labels, and the induced 2-dimensional dynamic graph $G^2$.

1.2 What Follows

Of the models presented in the preceding section, it appears that only the dynamic graph model lends itself to efficient processing. However, dynamic graphs are infinite graphs, which limits the type of problems to which they are applicable.

We focus our attention on the hierarchical graph model of Lengauer [19]. Any graph can be described by a hierarchical graph, although not necessarily at a large savings in space. However, there are infinite families of graphs whose sizes are exponentially larger than their hierarchical descriptions (see Section 2.2.2). As we shall see, many problems that have efficient non-hierarchical algorithms (i.e., algorithms that operate on non-hierarchical graphs), also have efficient hierarchical algorithms.

The remainder of this dissertation is organized as follows:
• In Chapter 2, we review graph theory and introduce the hierarchical graph model.

• In Chapter 3, we consider methods for processing hierarchical graphs. We illustrate Lengauer's bottom-up method with a hierarchical algorithm for bipartiteness testing, and then generalize the crucial properties of the bottom-up method in order to make it amenable to a wider variety of problems.

• In Chapter 4, we consider connectivity augmentation problems. These are problems of determining the minimum number of edges which must be added to a
graph to make it satisfy some particular connectivity property. We develop polynomial-time hierarchical algorithms for bridge-connectivity, biconnectivity, and strong-connectivity augmentation. A preliminary version of this work appears in [9].

- Two subgraph homeomorphism problems are considered in Chapter 5. In particular, we develop linear-time hierarchical algorithms for testing whether the expansion of a hierarchical graph is series-parallel [7], or outer-planar [3]. These classes of graphs can both be characterized as those graphs that contain no subgraph homeomorphic from one of a finite number of forbidden graphs. We also describe a polynomial-space hierarchical algorithm for extracting a forbidden subgraph from the expansion of a hierarchical graph, provided one exists.

- In Chapter 6, we develop a hierarchical greedy algorithm for finding bases of certain classes of matroids defined on the edges of graphs. Using this algorithm, we develop polynomial-time hierarchical algorithms for computing costs of optimum bases of matroids from two infinite families of matroids. We also develop polynomial-space hierarchical algorithms for generating an optimum base instead of its cost. Portions of this work appear in [10].

- In Chapter 7 we summarize the results presented, examine some common characteristics of the problems solved, and discuss some open problems.
2. PRELIMINARIES

2.1 Graph Theory

In this section, we introduce the set and graph-theoretical notation, terminology, and concepts used throughout the remaining chapters.

Let $S$ and $T$ be sets. We write $e \in S$ to indicate that $e$ is an element of $S$. The cardinality of $S$ is denoted by $|S|$. The empty set is written $\emptyset$. $S \subseteq T$ indicates that $S$ is a subset of $T$, and $S \subset T$ indicates proper inclusion. The union and intersection of $S$ and $T$ are written $S \cup T$ and $S \cap T$, respectively. $S - T$ consists of those elements of $S$ that are not in $T$. The power set of $S$, denoted $\mathcal{P}(S)$, is the set of all subsets of $S$. $S \times T$, the Cartesian product of $S$ and $T$, is the set of all ordered pairs $(a, b)$ such that $a \in S$ and $b \in T$. A binary operation $\ast$ on $S$ is a mapping from $S \times S$ into $S$.

We also use the standard logical connectives: and ($\land$), or ($\lor$), implication ($\implies$), and equivalence ($\iff$).

2.1.1 Basic terminology and concepts

A graph $G$ consists of a set $V(G)$ of vertices, and a multi-set $E(G)$ of edges. Each edge consists of a pair of vertices called its endpoints. $E(G)$ is a multi-set because we allow distinct edges to have the same endpoints. Hereafter, we shall refer to $E(G)$ simply as a set. We say $e = (u, v)$ is incident on $u$ and $v$, and $u$ and $v$ are adjacent. In an undirected graph, $(u, v)$ and $(v, u)$ are identical. In a directed graph,
edge \((u, v)\), usually called an arc, is directed from \(u\) to \(v\), and therefore, is different than arc \((v, u)\). For an arc \((u, v)\), \(u\) is called its tail and \(v\) its head. Unless otherwise specified, all graphs are assumed to be undirected.

A loop is an edge whose endpoints are not distinct. Two distinct edges having the same endpoints are said to be parallel edges, and an edge is redundant if it is parallel to another edge. To distinguish between edges in parallel, we assume that every edge has a name that is independent of its endpoints. A graph is simple if it has no redundant edges and no loops. We assume graphs may not be simple, and we use the term graph in this more general sense.

Let \(v \in V(G)\). The degree of \(v\) in \(G\), denoted \(\deg_G(v)\), is the number of times \(v\) appears as an endpoint of an edge. Thus, every loop incident on \(v\) contributes 2 to \(\deg_G(v)\). When the identity of the graph \(G\) in question is clear, we will use \(\deg(v)\) to denote \(\deg_G(v)\). A vertex \(v\) is isolated if \(\deg(v) = 0\), is a leaf if \(\deg(v) = 1\), and is a series vertex if \(\deg(v) = 2\). A pendant edge is an edge incident on a leaf. A loop is isolated if its endpoint is a series vertex. A series edge is an edge incident on a series vertex, and two edges are in series if they have a common endpoint that is a series vertex.

A path is a sequence \(P = v_1, v_2, \ldots, v_n\) of \(n \geq 1\) vertices such that \((v_i, v_{i+1}) \in E(G)\) for each \(1 \leq i < n\). Vertices \(v_1\) and \(v_n\) are connected by \(P\). The length of \(P\) is its number of edges. \(P\) is a simple path if it contains no repeated vertices, a closed path if \(v_1 = v_n\), and a cycle if it is closed and all vertices except \(v_1\) and \(v_n\) are distinct. If \(G\) is directed, then each \((v_i, v_{i+1})\) is directed from \(v_i\) to \(v_{i+1}\), and \(P\) is called a directed path.

A simple graph is a forest if it contains no cycles, and is a tree if it is a forest in which every two vertices are connected by a path. A directed acyclic graph (dag) is
a directed graph having no directed cycles. Let $G$ be a simple graph. $G$ is complete if every two vertices are adjacent. The complete graph on $n$ vertices is denoted $K_n$. If $V(G)$ can be partitioned into two sets $S_1$ and $S_2$ such that every edge in $G$ has one endpoint in $S_1$ and the other in $S_2$, then $G$ is bipartite, and $S_1$ and $S_2$ are a bipartition of $V(G)$. $G$ is a complete bipartite graph if each vertex in $S_1$ is adjacent to every vertex in $S_2$. The complete bipartite graph whose bipartition contains $n$ and $m$ vertices is denoted by $K_{n,m}$.

2.1.2 Subgraphs, minors, and homeomorphism

Let $G$ be a graph. A graph $H$ is a subgraph of $G$, denoted $H \subseteq G$, provided $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Let $S \subseteq V(G)$. $G - S$ is the graph with vertex set $V(G) - S$, and edge set $\{e \in E(G) : \text{no endpoint of } e \text{ belongs to } S\}$. For brevity, we denote $G - \{v\}$ by $G - v$.

Let $S \subseteq E(G)$. $G - S$ is a graph with vertex set $V(G)$ and edge set $E(G) - S$. $G/S$ is a graph obtained from $G$ by contracting each edge in $S$. An edge $e = (u, v) \in E(G)$ is contracted by deleting $e$ and identifying its endpoints, a process that may introduce parallel edges. The resulting vertex is denoted $uv$ or, equivalently, $vu$. However, we occasionally find it useful to assume that one of the endpoints of $e$, for instance $u$, is a vertex of $G/e$. In this case, we say $v$ was identified with $u$, and $e$ was contracted into $u$. Note that the graph obtained by contracting $e$ into $u$ is isomorphic to the graph obtained by contracting $e$ into $v$. The proof of the following fact is straightforward:

**Fact 2.1.1** For any graph $G$ and any disjoint subsets $S$ and $T$ of $E(G)$, $(G - S) - T = G - (S \cup T)$, $(G/S)/T = G/(S \cup T)$, and $(G - S)/T = (G/T) - S$.

A minor of $G$ is obtained from a subgraph of $G$ by applying a sequence of edge contractions. Then, disregarding isolated vertices, we have by Fact 2.1.1 that any
minor of $G$ can be written as $(G - D)/C$ for some disjoint subsets $C$ and $D$ of $E(G)$. Figure 2.1 shows a graph and some of its minors.

Let $u, v \in V(G)$. $G + (u, v)$ is obtained from $G$ by adding an edge with endpoints $u$ and $v$, possibly creating a redundant edge. If $s \notin V(G)$, $G + \{(u, s), (s, v)\}$ denotes the result of adding $s$ and edges $(u, s)$ and $(s, v)$ to $G$. An edge $e = (u, v)$ of $G$ is subdivided by creating the graph $(G - e) + \{(u, s), (s, v)\}$, where $s \notin V(G)$. $H$ is homeomorphic from $G$, denoted $G \leq H$, if $H$ can be obtained from $G$ by edge subdivisions. $G \subseteq H$ indicates that $H$ contains a subgraph homeomorphic from $G$. $H$ is homeomorphic to $G$ if $H$ and $G$ are homeomorphic from a common graph $J$.

2.1.3 Graph connectivity

In this section, we review the notions of connectivity, biconnectivity, bridge-connectivity, strong connectivity (for directed graphs), and triconnectivity.

A graph $G$ is connected if every two vertices are connected by a path. A connected component of $G$ is a maximally connected subgraph of $G$. A vertex $v$ of $G$ is a cutpoint if $G - v$ has more connected components than $G$. $G$ is said to be biconnected if it is connected and has no cutpoints. A biconnected component (block) of $G$ is a maximally biconnected subgraph of $G$. 
Definition 2.1.1 [8,22] The biconnectivity forest (bc-forest) of a graph $G$, denoted $bcf(G)$, is a forest with two types of nodes: b-nodes and c-nodes. Corresponding to each b-node $b$ is a unique block $\beta(b)$ of $G$, and corresponding to each c-node $c$ is a unique cutpoint $\kappa(c)$ of $G$. Two nodes $u$ and $v$ of $bcf(G)$ are adjacent if and only if $u$ is a b-node, $v$ is a c-node, and $\kappa(v) \in V(\beta(u))$.

The blocks, cutpoints, and bc-forest of $G$ can be found in linear time [34]. Figure 2.2 shows a graph and its bc-forest. Circles denote b-nodes and triangles denote c-nodes.

An edge $e$ of $G$ is a bridge if and only if $G - e$ has more connected components than $G$. Equivalently, a bridge is an edge that belongs to no cycles. $G$ is bridge-connected if it is connected and has no bridges. A bridge-connected component of $G$ is a maximally bridge-connected subgraph of $G$. The set of all biconnected graphs is properly contained in the set of all bridge-connected graphs.

Definition 2.1.2 [8] The bridge-connectivity forest (br-forest) of a graph $G$, denoted $brf(G)$, is a forest in which each node $v$ corresponds to a unique bridge-connected component $\beta(v)$ of $G$. Nodes $u$ and $v$ of $brf(G)$ are adjacent if and only if there is a
Figure 2.3: Graph $G$ and its br-forest $\text{brf}(G)$

bridge $(x, y)$ of $G$ such that $x \in V(\beta(u))$ and $y \in V(\beta(v))$.

The br-forest of $G$ can be constructed in linear time [8]. Figure 2.3 shows a graph $G$ and its br-forest.

A directed graph $G$ is strongly-connected if $G$ has a directed path from each vertex to every other vertex. A strongly-connected component of $G$ is a maximally strongly-connected subgraph of $G$.

Definition 2.1.3 [8] The strong-connectivity dag (st-dag) of a directed graph $G$, denoted $\text{std}(G)$, is a directed acyclic graph in which each node $v$ corresponds to a unique strongly-connected component $\beta(v)$ of $G$. Arc $(u, v)$ belongs to $\text{std}(G)$ if and only if there is an arc $(x, y) \in E(G)$ such that $x \in V(\beta(u))$ and $y \in V(\beta(v))$. A source of $\text{std}(G)$ is a vertex with no incoming arcs, and a sink of $\text{std}(G)$ is a vertex with no outgoing arcs.

The st-dag of $G$ can be constructed in linear time [8]. Figure 2.4 shows a directed graph and its st-dag.

The following definitions of triconnectivity and triconnected components are from Hopcroft and Tarjan [15]. Let $G$ be a biconnected graph and $S = \{a, b\}$ a
Figure 2.4: Directed graph $G$ and its st-dag $\mathrm{std}(G)$

pair of vertices of $G$. Partition $E(G)$ into equivalence classes $E_1, E_2, \ldots, E_m$, called separation classes, such that two edges are in the same class if and only if they lie on a common path in $G$ containing no vertex of $S$ except as an endpoint. If $m \geq 2$, the pair $S$ is called a separation pair unless (1) $m = 2$ and one class contains a single edge, or (2) $m = 3$ and each class contains a single edge. $G$ is triconnected if it is biconnected and contains no separation pairs.

Let $S = \{a, b\}$ be a separation pair of $G$. Split $G$ into graphs $G_1$ and $G_2$, each having at least two edges, such that $E(G_i)$ is the union of some of the separation classes of $G$, $E(G_1) \cap E(G_2) = \emptyset$, $V(G_1) \cap V(G_2) = S$, and $V(G_1) \cup V(G_2) = V(G)$. Let $G_i'$ be obtained by adding virtual edge $(a, b)$ to $G_i$. The two virtual edges are called a companion pair, and we say that $G_1'$ and $G_2'$ share the companion pair. The graphs $G_1'$ and $G_2'$ are called split graphs of $G$ with respect to $S$. The splitting process is continued separately on $G_1'$ and $G_2'$ until no more splits are possible. The resulting graphs are called the split components of $G$. The split components, which may not be unique, have at most $3 \cdot |E(G)| - 6$ edges, and can be constructed in $O(|V(G)| + |E(G)|)$ time [15]. Each split component is a triconnected graph, a triangle, or a triple bond (two vertices connected by three parallel edges).

Two split components sharing a companion pair are merged by deleting the
companion edges and appropriately identifying their endpoints. The *triconnected components* of $G$ are unique [15], and are obtained by merging all polygons (i.e., cycles) that share companion pairs, and merging all bonds that share companion pairs. Thus, the triconnected components of $G$ consist of triconnected graphs, polygons, and multiple bonds. Each edge of $G$ belongs to exactly one triconnected component, and is called a *real* edge of that component.

**Theorem 2.1.1** [15] The triconnected components of a biconnected graph $G$ have at most $3 \cdot |E(G)| - 6$ edges, and can be found in $O(|V(G)| + |E(G)|)$ time.

**Definition 2.1.4** [15] Let $G$ be a biconnected graph. The *triconnectivity tree* of $G$, denoted $tct(G)$, is a tree in which each vertex $v$ corresponds to a unique triconnected component $\beta(v)$ of $G$. Two nodes $u$ and $v$ of $tct(G)$ are adjacent if and only if $\beta(u)$ and $\beta(v)$ share a companion pair.

Given Theorem 2.1.1, $tct(G)$ can be constructed in linear time. Figure 2.5 shows a biconnected graph $G$, its triconnected components, and its triconnectivity tree. Virtual edges are indicated by dashed lines, and each companion pair is identified by an upper-case letter positioned between its two virtual edges.

### 2.2 The Hierarchical Graph Model

In this section, we introduce the hierarchical graph model, and determine bounds on the size of the expansion of a hierarchical graph in terms of the size of the hierarchical graph. We conclude the section with a brief discussion of previous work on hierarchical graphs.
Figure 2.5: (a) $G$, (b) the triconnected components of $G$, and (c) $tct(G)$
2.2.1 Hierarchical graphs

The following concepts and definitions are adapted from [20,22,24]. We assume that every graph $G$ has zero or more distinguished vertices called coupling vertices. All other vertices are called non-coupling vertices. The coupling vertices of $G$ are used to glue $G$ onto other graphs as follows:

**Definition 2.2.1** Let $H$ be a graph, and let $L = [(a_1, b_1), \ldots, (a_r, b_r)]$ be a list of pairs where $a_1, \ldots, a_r$ are distinct (not necessarily coupling) vertices of $H$, and $b_1, \ldots, b_r$ are the distinct coupling vertices of $G$. $L$ is called a gluing list, and $H$ and $L$ are said to be compatible with $G$. The graph $J = H \circ_L G$ is the result of gluing $G$ onto $H$ using $L$. If $L$ is empty, $J$ is simply the disjoint union of $G$ and $H$. Otherwise, $J$ is obtained by identifying $a_i$ and $b_i$ for $1 \leq i \leq r$. The coupling vertices of $J$ are those of $H$. We will omit gluing lists in situations where no confusion can arise.

If for $i = 1, \ldots, m$, the graph $G_i$ is to be glued onto $H$ via list $L_i$, the result is written $(\cdots (((H \circ_{L_1} G_1) \circ_{L_2} G_2) \circ_{L_3} \cdots \circ_{L_m} G_m))$. When parentheses are omitted, we assume that the operation is left associative. Again, if no confusion can arise, we shall not display the gluing lists.

For any graph $G$, let $\text{Env}(G)$, the environment of $G$, be the set of all graph and gluing list pairs $(H, L)$ such that $H$ and $L$ are compatible with $G$. Then, two graphs have the same environment if and only if they have the the same coupling vertices. The following properties are consequences of Definition 2.2.1.

**Lemma 2.2.1** Let $G$ and $H$ be graphs, and suppose $(H, L) \in \text{Env}(G)$. Then,

1. $\text{Env}(H \circ_L G) = \text{Env}(H)$.
2. If $(H, N) \in \text{Env}(F)$, then $(H \circ_L G, N) \in \text{Env}(F)$ and

$(H \circ_L G) \circ_N F = (H \circ_N F) \circ_L G$. 
3. If \((G, N) \in \text{Env}(F)\), then \((H \circ_L G, N) \in \text{Env}(F)\) and 
\[(H \circ_L G) \circ_N F = H \circ_L (G \circ_N F).\]

4. There exists some \((J, N) \in \text{Env}(G)\) such that for any \(F\) for which 
\[\text{Env}(F) = \text{Env}(G),\] 
\[J \circ_N F = F.\]

**Proof:** Item (1) is a consequence of the assumption that the coupling vertices of 
\(H \circ_L G\) are the coupling vertices of \(H\). Item (2) is straightforward given that the 
vertices in a gluing list are distinct.

Consider item (3). Since the vertices in a gluing list are distinct, \(H \circ_L G\) contains 
a subgraph \(G'\) isomorphic to \(G\), although the vertex names of \(G'\) may different than 
those of \(G\). However, the correspondence between vertices of \(G'\) and \(G\) is clear. By 
acknowledging this correspondence, \((H \circ_L G) \circ_N F\) is well defined.

Consider item (4). Suppose the coupling vertices of \(G\) are named \(v_1, v_2, \ldots, v_r\). 
Let \(V(J) = \{v_1, \ldots, v_r\}\), \(E(J) = \emptyset\), and \(N = [(v_1, v_1), \ldots, (v_r, v_r)]\). Then \((J, N) \in \text{Env}(G)\), and 
\(J \circ_L F = F\) for any \(F\) such that \(\text{Env}(F) = \text{Env}(G)\). □

Let \(s\) and \(t\) be strings. Denote by \(s \cdot t\) the concatenation of \(s\) and \(t\) where the 
division between \(s\) and \(t\) is not lost. We extend this notion to a *prefix* operator on 
graphs and sets as follows. For a set \(A\), \(s \cdot A = \{s \cdot a : a \in A\}\). For a graph \(G\), \(s \cdot G\) 
is a graph with vertex set \(s \cdot V(G)\) and edge set \(s \cdot E(G)\).

**Definition 2.2.2** A *hierarchical graph* \(\Gamma = (G_1, \ldots, G_n)\) is a finite list of simple 
graphs called *cells*. For each \(i\), \(V(G_i)\) is partitioned into *pins*, *terminals*, and *nonterminals*. \(G_i\) has \(p_i\) pins whose names are the integers \(1, \ldots, p_i\). The coupling vertices 
of \(G_i\) are its pins. Each nonterminal of \(G_i\) has a name of the form \(n \cdot t\), where \(n\) is 
an integer, and \(t\), its *type*, is a symbol in the set \(\{G_1, \ldots, G_{i-1}\}\). A nonterminal of
type $G_j$ has degree $p_j$, and each incident edge is labeled with the name of a unique pin of $G_j$. Edges between nonterminals are not permitted.

**Definition 2.2.3** Let $\Gamma = (G_1, \ldots, G_n)$ be a hierarchical graph. Associated with $\Gamma$ is a hierarchy tree $T$. Each node of $T$ is labeled with the name of a cell of $\Gamma$. The root has label $G_n$. Let $v$ be a node of $T$ with label $G_i$, and assume the nonterminals of $G_i$ are $1 \cdot G_{i_1}, \ldots, m \cdot G_{i_m}$. Then, $v$ has $m$ children $v_1, \ldots, v_m$ where $v_j$ has label $G_{i_j}$, and edge $(v, v_j)$ has label $j$. A pathname is a sequence $l_1 \cdot l_2 \cdot \ldots \cdot l_r$ consisting of the edge labels on a path in $T$ from the root to some node in $T$.

Let $G_i$ and $G_j$ be cells of $\Gamma$, and suppose $G_i$ contains a nonterminal $v = l \cdot G_j$. Let $v(1), \ldots, v(p_j)$ be the vertices of $G_i$ adjacent to $v$ such that for each $m$, edge $(v, v(m))$ has label $m$. We replace the nonterminal $v$ with the graph $l \cdot G_j$ by forming the composition $(G_i - v) \circ L \cdot G_j$ where $L = [(v(1), 1), \ldots, (v(p_j), p_j)]$. The pins of the resulting graph are those of $G_i$.

**Definition 2.2.4** Let $\Gamma = (G_1, \ldots, G_n)$ be a hierarchical graph. For $1 \leq i \leq n$, $\Gamma_i = (G_1, \ldots, G_i)$ is a hierarchical graph describing the graph $X(\Gamma_i)$, the expansion of $\Gamma_i$, which is defined recursively as follows:

- If $G_i$ has no nonterminals, then $X(\Gamma_i) = G_i$.
- If $G_i$ has nonterminals, then $X(\Gamma_i)$ is obtained from $G_i$ by replacing each nonterminal $l \cdot G_j$ by the graph $l \cdot X(\Gamma_j)$.

We denote $X(\Gamma_n)$ by $X(\Gamma)$. Note that $X(\Gamma)$ may have redundant edges even though each of its cells is simple. The name of each edge (vertex) of $X(\Gamma_i)$ consists of a pathname in the hierarchy tree of $\Gamma_i$ followed by the name of an edge (vertex) in some cell of $\Gamma_i$. We shall assume that $\Gamma_n = \Gamma$ has no useless cells — cells that
are not the type of any nonterminal in $\Gamma$. It is clear that if $\Gamma$ had useless cells, they could easily be removed in $O(|\Gamma|)$ time. Figure 2.6 shows a hierarchical graph with three cells, and the graphs $X(\Gamma_i)$, $1 \leq i \leq 3$. Pins are represented by squares, nonterminals by large circles, and terminals by small circles. Figure 2.7 gives the hierarchy tree of $\Gamma$ from Figure 2.6.
2.2.2 Comparing the sizes of $\Gamma$ and $X(\Gamma)$

Let the size of a graph $G$ be defined as $|G| = |V(G)| + |E(G)|$. We define the size of a hierarchical graph $\Gamma$, written $|\Gamma|$, to be the sum of the sizes of its cells. In this section, we show that for any hierarchical graph $\Gamma$, the size of $X(\Gamma)$ is at most exponential in the size of $\Gamma$. We also describe an infinite family of hierarchical graphs in which exponential blow-up is achieved.

Theorem 2.2.1 Let $\Gamma = (G_1, G_2, \ldots, G_n)$. Then, $|X(\Gamma)| \leq 2^{|\Gamma|}$.

Proof (induction on $n$): Assume $G_i$ has $p_i$ pins, $t_i$ terminals, $e_i$ edges, and $r_i$ nonterminals. For each $j < i$, let $r_{i,j}$ be the number of nonterminals of $G_i$ that have type $G_j$. Let $S_i = |\Gamma_i|$. We must show $|X(\Gamma)| \leq 2^S$. If $n = 1$, then $X(\Gamma) = G_1$, and the result follows. Assume $n > 1$. Then, $r_n \geq 1$ since $\Gamma$ is assumed to have no useless cells.

When a nonterminal $v = l \cdot G_j$ of $G_i$ is replaced by $X(\Gamma_j)$, $v$ and its $p_j$ incident edges are deleted, and each of the $p_j$ pins of $X(\Gamma_j)$ is identified with a vertex of $G_i$. Thus, the resulting graph has size $e_i + (r_i - 1) + p_i + t_i + |X(\Gamma_j)| - 2 \cdot p_j$. Therefore, we have the following:
\[|X(\Gamma)| \leq e_n + p_n + t_n + \sum_{i=1}^{n-1} r_{n,i} \cdot |X(\Gamma_i)|\]

\[\leq e_n + p_n + t_n + \sum_{i=1}^{n-1} r_{n,i} \cdot 2^{S_i} \quad \text{by the induction hypothesis}\]

\[\leq e_n + p_n + t_n + r_n \cdot \sum_{i=1}^{n-1} 2^{S_i} \quad \text{since } r_n \geq r_{n,i} \text{ for } i < n\]

\[\leq e_n + p_n + t_n + r_n \cdot 2^{(1+S_{n-1})} \quad \text{since } S_{i+1} > S_i \text{ for each } i\]

\[= e_n + p_n + t_n + 2^{(\log r_n + 1 + S_{n-1})} \quad \text{since } r_n \geq 1\]

\[\leq e_n + p_n + t_n + 2^{(r_n + S_{n-1})} \quad \text{since } \log r_n \leq r_n - 1\]

\[\leq 2^{(e_n + p_n + t_n + r_n + S_{n-1})}\]

\[= 2^{S_n} \quad \square\]

By the preceding theorem, at most exponential blow-up is possible. We now show that there are families of hierarchical graphs in which exponential blow-up occurs.

Consider the infinite family of hierarchical graphs \(\mathcal{F} = \{\Gamma^1, \Gamma^2, \Gamma^3, \ldots\}\), where for each \(n > 0\), \(\Gamma^n = (G_1, \ldots, G_n)\) is of the form:

- \(G_1: \quad \text{O} \quad \text{I} \quad G_i (1 < i \leq n): \quad \begin{array}{cc} & 1 \\text{G}_{i-1} & \text{G}_{i-1} \\text{G}_{i-1} \end{array} \begin{array}{cc} 1 \end{array} \)

Then, \(|\Gamma^n| = |E(\Gamma^n)| + |V(\Gamma^n)| = 2(n - 1) + (2 + 4(n - 1)) = 6n - 4\). The hierarchy tree of \(\Gamma^n\) is a complete binary tree of depth \(n\), and hence, has exactly \(2^n - 1\) nodes. Thus, given the description of \(\Gamma^n\), it follows that \(|E(X(\Gamma^n))| = 0\) and \(|V(X(\Gamma^n))| = 2^n\). Therefore, \(|X(\Gamma^n)| \geq 2^{|\Gamma^n|/6}\). How close to 1 the constant of proportionality in the exponent can be seems to be an open problem.
2.2.3 Previous results on hierarchical graphs

Lengauer and Wanke [24] provided linear-time hierarchical algorithms to determine whether the expansion of a hierarchical graph is connected or biconnected. They also described a polynomial-time hierarchical strong-connectivity testing algorithm. In addition, they showed how to construct from $\Gamma$ a hierarchical graph $\Gamma'$ such that $X(\Gamma') = X(\Gamma)$, and $\Gamma'$ explicitly describes the connected or biconnected components of $X(\Gamma)$.

In [19], Lengauer considered problems related to minimum cost spanning trees (MSTs). He developed a hierarchical algorithm that determines the cost of an MST of $X(\Gamma)$ in time almost linear in the size of $\Gamma$. The running time is dominated by the running time of any non-hierarchical MST algorithm. Lengauer also showed how to generate the edges in an MST of $X(\Gamma)$ using work space linear in the size of $\Gamma$, and described efficient methods for answering certain types of queries on an MST of $X(\Gamma)$.

A planar graph is a graph that can be embedded in the plane so that no edges intersect except at their endpoints. Lengauer [20] developed linear-time hierarchical algorithms for determining whether or not $X(\Gamma)$ is planar or strongly planar. $X(\Gamma)$ is strongly planar if it has a planar embedding in which the pins of all cell instances occur around the “boundary” of the cell instance. Lengauer also outlined algorithms for constructing planar and strongly planar embeddings of $X(\Gamma)$.

Clearly, many problems that have polynomial-time non-hierarchical algorithms also have polynomial-time hierarchical algorithms. However, this is not always the case. For instance, the Circuit Value Problem (CVP) and Network Flow have polynomial-time non-hierarchical solutions, but were shown to be PSPACE-complete under the hierarchical model by Lengauer and Wagner [21]. The difficulty with CVP
is that the output of a circuit does not depend solely on its structural properties. The output also depends on the input, and on the function computed by each logic gate in the circuit. Network flow suffers from a similar problem. Lengauer and Wagner [21] also showed that, in general, there is no correlation between the hierarchical and non-hierarchical complexities of graph problems.
3. PROCESSING HIERARCHICAL GRAPHS

Determining a property of $X(T)$ using polynomial-time in the size of $T$ requires a method that somehow exploits the regular structure of $X(T)$. The bottom-up method (BU-method) of Lengauer and Wanke [24] performs this task particularly well on many decision problems (problems with yes or no answers). In Section 3.1 we illustrate the BU-method with a hierarchical algorithm for bipartiteness testing.

Not every graph problem has a simple yes or no answer. With the addition of the BU-table [19,24], the BU-method can be applied to non-decision problems. However, we would prefer a more general methodology that fits many different types of problems. We achieve this goal by generalizing certain properties crucial to the correctness of the bottom-up method. We conclude Section 3.1 with an example illustrating our generalized bottom-up method, and then present the method in detail in Section 3.2.

Many of the algorithms we present in the coming chapters use simple arithmetic to compute quantities that depend on the size of the expansion of a hierarchical graph. We conclude this chapter with a discussion of the complexity of such arithmetic operations.
3.1 The Bottom-Up Method

Consider the decision problem $\Pi(G) = \text{"is } G \text{ bipartite?"}$. Given $\Gamma$, the goal is to determine $\Pi(X(\Gamma))$ as efficiently as possible. The obvious procedure is to simply construct $X(\Gamma)$ and apply some non-hierarchical algorithm for $\Pi$. However, this approach may be impractical because of the potential difference in size between $\Gamma$ and $X(\Gamma)$. The BU-method attempts to exploit the regularity of $X(\Gamma)$, a property that is expressed in its hierarchical description. In this section, we describe the BU-method, and then present some techniques that can be used to extend the BU-method to a wider range of problems.

3.1.1 Algorithm BU

The BU-method is implemented by algorithm BU. We illustrate algorithm BU with an algorithm that, given $\Gamma$, determines $\Pi(X(\Gamma))$ (i.e., whether or not $X(\Gamma)$ is bipartite) in time linear in the size of $\Gamma$. Recall the definition of a bipartite graph. $G$ is bipartite if and only if $V(G)$ can be partitioned into two sets such that no two vertices within the same set are adjacent. Such a partition is called a bipartition of $V(G)$. Equivalently, $G$ is bipartite if and only if $G$ contains no cycles of odd length.

The correctness of the bottom-up method relies on replaceability, a property defined as follows:

Definition 3.1.1 [24] Graphs $G$ and $G'$ are said to be replaceable with respect to $\Pi$, written $G \approx_{\Pi} G'$, if and only if $\text{Env}(G) = \text{Env}(G')$ and, for every $(H, L) \in \text{Env}(G)$, $\Pi(H \circ_L G) = \Pi(H \circ_L G')$.

Replaceability is an equivalence relation on graphs. Note that by Lemma 2.2.1, $G \approx_{\Pi} G'$ implies $\Pi(G) = \Pi(G')$. The BU-method attempts to quickly find a "small"
Algorithm BU

input: $\Gamma = (G_1, G_2, \ldots, G_n)$
output: $\Pi(X(\Gamma))$

for $i = 1$ to $n$ do
  if $G_i$ contains no nonterminals then
    $\bar{G}_i := G_i$
  else
    Build $\bar{G}_i$ by replacing each nonterminal $l \cdot G_j$ of $G_i$ with $l \cdot G_j^b$;
    $G_i^b := B(\bar{G}_i)$;
  end;
return $\Pi(G_i^b)$

Figure 3.1: Algorithm BU

graph that is replaceable with $X(\Gamma)$. Finding replaceable graphs is the responsibility of a function $B$ called the burner [22]. The burner takes a graph $G$ as input and returns a burnt graph $B(G)$, denoted $G^b$, such that $G^b \approx \Pi G$.

Applied to $\Gamma = (G_1, \ldots, G_n)$, algorithm BU constructs sequences of graphs $\bar{G}_1, \bar{G}_2, \ldots, \bar{G}_n$ and $G_1^b, G_2^b, \ldots, G_n^b$ such that for each $i$, $G_i^b \approx \Pi \bar{G}_i \approx \Pi X(\Gamma_i)$. Thus, we can determine if $X(\Gamma)$ is bipartite by determining if $G_n^b$ is bipartite. We return to this issue in the next section. Algorithm BU is given in Figure 3.1, and the bipartiteness testing burner is given in Figure 3.2.

The name of each vertex (edge) of $\bar{G}_i$ and $G_i^b$ is a pathname in $\Gamma_i$ followed by the name of a vertex (edge) in some cell of $\Gamma_i$. A pathname in $\bar{G}_i$ consists of as many as $i$ symbols, each requiring $\log n$ space, where $n$ is the number of cells in $\Gamma$. Thus, the sizes of $\bar{G}_i$ and $G_i^b$ are determined by their number of vertices, edges, and the sizes of pathnames. However, for bipartiteness testing (and many of the problems
function $B(G: \text{graph})$:\text{graph}
begin
    if $G$ is not bipartite then
        Let $G^b$ consist of the pins of $G$ and a cycle of length 3
    else
        $G^b := G$;
        for each connected component $C$ of $G^b$ do
            Find bipartition $X, Y$ of $V(C)$;
            Let $P$ be the set of pins of $G$ in $C$;
            Delete $C$ from $G^b$;
            if $P \neq \emptyset$ then
                if $P \cap X = \emptyset$ or $P \cap Y = \emptyset$ then
                    Add to $G^b$ a bipartite graph consisting of a terminal vertex adjacent to each element of $P$
                else
                    Add to $G^b$ a connected bipartite graph with bipartition $P \cap X$ and $P \cap Y$
                end
            end
        end
    end
    return $G^b$
end

Figure 3.2: The bipartiteness testing burner
we shall solve), pathnames are not needed. Thus, we can save time and space if we eliminate pathnames by replacing each nonterminal $I^\ast G_j$ of $G_i$ with $G_j^b$ instead of $I^\ast G_j^b$. In Chapters 5 and 6 we shall encounter problems for which pathnames are necessary.

3.1.2 Correctness of algorithm BU

The correctness of algorithm BU for problem II depends upon the burner producing replaceable graphs.

Lemma 3.1.1 For any graph $G$, $B(G) \approx \Pi G$.

Proof: Let $(H, L) \in \text{Env}(G)$, and let $G^b = B(G)$. $G$ and $G^b$ have the same pins, and hence, the same environments. Since $H$ is arbitrary, we need only show that $\Pi(H \circ_L G) = \Pi(H \circ_L G^b)$. If $G$ is not bipartite, $G^b$ contains an odd length cycle, and hence, neither $H \circ_L G$ nor $H \circ_L G^b$ is bipartite.

Suppose $G$ is bipartite. Each connected component $C$ of $G$ is replaced by a connected bipartite graph $C^b$. By construction, two pins of $G$ are connected by an odd length path in $C$ if and only if they are connected by an odd length path in $C^b$. Thus, it follows that $H \circ_L G$ is bipartite if and only if $H \circ_L G^b$ is bipartite. □

The proof techniques used in the following three results are due to Lengauer and Wanke [22,24]. Notice that these results are properties of replaceability and graph gluing, and are independent of the problem II.

Lemma 3.1.2 If $G \approx \Pi G'$, then for any $(H, L) \in \text{Env}(G)$, $H \circ_L G \approx \Pi H \circ_L G'$.

Proof (contrapositive): By Lemma 2.2.1, $\text{Env}(H \circ_L G) = \text{Env}(H \circ_L G') = \text{Env}(H)$. Suppose $H \circ_L G$ and $H \circ_L G'$ are not replaceable. Then, there exists some $(J, N) \in$
Env(\(H\)) such that \(\Pi(J \circ_N (H \circ_L G)) \neq \Pi(J \circ_N (H \circ_L G'))\). But, by Lemma 2.2.1, \(J \circ_N (H \circ_L G) = (J \circ_N H) \circ_L G\) and \(J \circ_N (H \circ_L G') = (J \circ_N H) \circ_L G'\). Thus, \(\Pi((J \circ_N H) \circ_L G) \neq \Pi((J \circ_N H) \circ_L G')\). This implies that \(G\) and \(G'\) are not replaceable. □

The next lemma is the key result in the correctness of algorithm BU.

Lemma 3.1.3 Let \(\Gamma = (G_1, G_2, \ldots, G_n)\) be a hierarchical graph. Then, for \(1 \leq i \leq n\), \(G_i^b \approx_{\Pi} \bar{G}_i \approx_{\Pi} X(\Gamma_i)\).

Proof (induction on \(i\)):

\(i = 1\): Since \(G_1\) has no nonterminals, \(\bar{G}_1 = X(\Gamma_1) = G_1\). Then, by Lemma 3.1.1 and reflexivity, \(G_1^b = B(\bar{G}_1) \approx_{\Pi} \bar{G}_1 \approx_{\Pi} X(\Gamma_1)\).

\(i > 1\): By the induction hypothesis, \(G_j^b \approx_{\Pi} \bar{G}_j \approx_{\Pi} X(\Gamma_j)\) for each \(j < i\). Let the nonterminals of \(G_i\) be \(N_i = \{1 \cdot G_i, \ldots, m \cdot G_i \}_{m}\), and let and let \(G = G_i - N_i\).

For each \(1 \leq j \leq m\), let \(F_j = j \cdot G_i^b_j\) and \(H_j = j \cdot X(\Gamma_j)\). By construction, \(\bar{G}_i = G \circ F_1 \circ F_2 \circ \cdots \circ F_m\). By Definition 2.2.4, \(X(\Gamma_i) = G \circ H_1 \circ \cdots \circ H_m\). Consider any \(1 \leq j \leq m\). By Lemma 3.1.2 and repeated applications of Lemma 2.2.1, we have the following:

\[
G \circ H_1 \circ \cdots \circ H_n = G \circ H_1 \circ \cdots \circ H_{j-1} \circ H_{j+1} \circ \cdots \circ H_n \circ H_j
\approx_{\Pi} G \circ H_1 \circ \cdots \circ H_{j-1} \circ H_{j+1} \circ \cdots \circ H_n \circ F_j
= G \circ H_1 \circ \cdots \circ H_{j-1} \circ F_j \circ H_{j+1} \circ \cdots \circ H_n
\]

By applying the above transformation for each \(j\), it follows that \(\bar{G}_i \approx_{\Pi} X(\Gamma_i)\). Since \(G_i^b = B(\bar{G}_i)\), the result follows by Lemma 3.1.1. □

The correctness of algorithm BU can now be shown.
Theorem 3.1.1  \textit{Applied to hierarchical graph }\Gamma = (G_1, G_2, \ldots, G_n), \textit{algorithm BU returns }\Pi(X(\Gamma)).

\textbf{Proof:} By Lemma 3.1.3 and transitivity, \(G^b_n \cong_{\Pi} X(\Gamma)\). By Lemma 2.2.1, there is some \((H, L) \in \text{Env}(G_n^b)\) such that \(H \circ_L G^b_n = G^b_n\) and \(H \circ_L X(\Gamma) = X(\Gamma)\). Then, by Definition 3.1.1, \(\Pi(G^b_n) = \Pi(X(\Gamma))\). Therefore, algorithm BU returns \(\Pi(X(\Gamma))\).

\(\square\)

Algorithm BU consists of a burner that depends on the problem \(\Pi\) and the definition of replaceability, and a generic framework that invokes the burner. Lemma 3.1.2, Lemma 3.1.3, and Theorem 3.1.1 are properties of replaceability and graph gluing, and do not depend on \(\Pi\). Thus, to prove the correctness of an algorithm based on algorithm BU, one need only show that the burner preserves replaceability.

3.1.3 Run-time analysis

Although Theorem 3.1.1 establishes the correctness of the BU-method, the algorithm given is not useful if it does not save time or space over simply expanding \(\Gamma\) and computing \(\Pi(X(\Gamma))\) directly. First, we show that the bipartiteness testing burner has a linear-time implementation. Then, by analyzing the size of the burnt graphs, we show that algorithm BU operates in time linear in the size of \(\Gamma\).

Breadth-first search [2] can be used to determine in linear time whether a graph is bipartite, and if it is bipartite, find a bipartition of its vertices. If \(G\) is not bipartite, \(G^b\) consists simply of the pins of \(G\) and a cycle of length 3. Suppose \(G\) is bipartite. The burner examines each connected component \(C\) of \(G\) separately. Depth-first search [34] can be used to find the connected components in linear time. \(C\) is deleted if it contains no pins. Otherwise, the unique bipartition of \(C\) is found, and \(C\) is replaced by a connected bipartite graph \(C^b\) having at most \(p + 1\) vertices and \(p + 1\)
edges, where \( p \) is the number of pins in \( C \). Clearly, \( C^b \) can be constructed in time linear in the size of \( C \).

Suppose \( G \) contains \( p \) pins. By construction, \( G^b \) contains \( O(p) \) vertices and edges. Each nonterminal of \( G_i \) has as many incident edges as there are pins in the graph that replaces it. Thus, \( G_i \) has size linear in the size of \( G_i \), and can be constructed in time linear in the size of \( G_i \). Since the burner operates in linear time, the time required for the \( i \)th iteration of algorithm BU is \( O(|G_i|) \). Therefore, we have the following theorem.

**Theorem 3.1.2** Applied to hierarchical graph \( \Gamma \), algorithm BU determines whether \( X(\Gamma) \) is bipartite in \( O(|\Gamma|) \) time.

The preceding theorem applies to any problem \( \Pi \) for which a burner exists that preserves replaceability, operates in linear time, and produces graphs whose sizes are linear in their number of pins.

### 3.1.4 Extending the bottom-up method

We shall now illustrate an approach that allows us to apply the BU-method to a larger class of problems. Our approach is formalized in Section 3.2. Consider the following connectivity augmentation problem:

Given a graph \( G \), determine the number of additional edges necessary and sufficient to connect \( G \).

If \( G \) has \( m \) connected components, then \( m - 1 \) edges are necessary and sufficient to connect \( G \). Thus, our problem reduces to finding \( cn(G) \), the number of connected components of \( G \). More complex augmentation problems are considered in Chapter 4.
function $B_{cn}(R$: pair$):$pair$
begin$
    for each connected component $C$ of $R.G$ do
        if $C$ contains no pins then
            Delete $C$ from $R.G$;
            $R.m := R.m + 1$
        else
            Substitute $C^b$ for $C$ in $R.G$, where $C^b$ is a path whose vertices are the pins of $C$
        end
    end;
    return $R$
end

Figure 3.3: The connectivity augmentation burner

Our algorithm, which we call algorithm ConAug, uses a burner function denoted $B_{cn}$. ConAug and $B_{cn}$ manipulate pairs of the form $(G, m)$, where $G$ is a graph and $m$ is an integer. The burner adds to $m$ the number of connected components it removes from the graph $G$. We will often find it convenient to denote the graph and integer components of a pair $S$ by $S.G$ and $S.m$, respectively. $B_{cn}$, given in Figure 3.3, is a straightforward extension of a connectivity testing burner described by Lengauer and Wanke [22].

For a pair $S$, let $Env(S)$ be the the set of all ordered pairs $(R, L)$ such that $R$ is a pair, $L$ a gluing list, and $(R.G, L) \in Env(S.G)$. We extend the graph gluing operation to pairs by defining $R \circ_L S$ to be the pair $(R.G \circ_L S.G, R.m + S.m)$. Notice that pair composition satisfies properties analogous to those of graph gluing given in Lemma 2.2.1.
Algorithm ConAug

input: $\Gamma = (G_1, \ldots, G_n)$
output: $\text{cn}(X(\Gamma)) - 1$

for $i = 1$ to $n$ do
    if $G_i$ contains no nonterminals then
        $\bar{T}_i := (G_i, 0)$
    else
        Let $G$ be the result of deleting all nonterminals from $G_i$;
        $\bar{T}_i := (G, 0)$;
        for each nonterminal $l \cdot G_j$ of $G_i$ do
            $\bar{T}_i := \bar{T}_i \circ T_j^b$
        end;
        $T_i^b := \text{Bcn}(\bar{T}_i)$
    end;
return $\text{cn}(T_n^b.G) + T_n^b.m - 1$

Figure 3.4: Algorithm ConAug

Let $\Gamma = (G_1, \ldots, G_n)$ be a hierarchical graph. Algorithm ConAug, given in Figure 3.4, constructs sequences of pairs $\bar{T}_1, \bar{T}_2, \ldots, \bar{T}_n$ and $T_1^b, T_2^b, \ldots, T_n^b$. It can be shown that for each $i$, $\text{cn}(T_i^b.G) + T_i^b.m = \text{cn}(\bar{T}_i.G) + \bar{T}_i.m = \text{cn}(X(\Gamma_i))$, and hence, that algorithm ConAug is correct. We postpone proving this result until after we have presented our generalized BU-method. Instead, we shall show that ConAug runs in time linear in the size of the hierarchical graph $\Gamma$.

Let $R$ be any pair. The burner considers each connected component $C$ of $R.G$ separately. Connected components can be found in linear time [34]. If $C$ contains no pins, $\text{Bcn}$ increments $R.m$ and deletes $C$ from $R.G$. Otherwise, $\text{Bcn}$ substitutes a graph $C^b$ for $C$ in $R.G$, where $C^b$ consists of a path whose only vertices are the
pins of $C$. Thus, $B(R).G$ has $p$ vertices and at most $p - 1$ edges, where $p$ is the number of pins of $R.G$. It follows that $|\overline{T}_i.G|$ is $O(|G_i|)$ (in fact, $\overline{T}_i.G$ has fewer vertices and fewer edges than $G_i$), and $\overline{T}_i.G$ can be constructed in $O(|G_i|)$ time. Furthermore, the total number of additions involved in computing $\overline{T}_i.m$ and $T_i^b.m$ is $O(|G_i|)$. We determine the cost of the arithmetic performed using the unit cost model [1], which assigns one unit of cost to each addition (a discussion of this choice appears in Section 3.3). Therefore, the $i$th iteration of ConAug takes $O(|G_i|)$ time, and hence, ConAug operates in $O(|\Gamma|)$ time.

3.2 The Generalized Bottom-Up Method

We now formalize the techniques used in algorithm ConAug in order to describe a more general approach to hierarchical processing that we call the generalized BU-method (GBU-method).

This section is organized as follows. First, we define gluing systems, which are algebraic structures that satisfy the graph-gluing properties given in Lemma 2.2.1. Using gluing systems, we define a generalization of hierarchical graphs called hierarchical objects, and an abstract notion of problems on hierarchical objects. Next, we present algorithm GBU, an algorithm for solving problems on hierarchical objects, and prove its correctness. Algorithm GBU implements the GBU-method. We conclude the section by using the GBU-method to prove the correctness of algorithm ConAug from Section 3.1.4.

3.2.1 Gluing systems and hierarchical objects

Our first goal is to capture the algebraic properties of graph gluing necessary to the correctness of the bottom-up method.
Definition 3.2.1 Let $\mathcal{O}$ be a nonempty set of objects, and $\star$ a binary operation on $\mathcal{O}$ that need not be defined for every pair of elements from $\mathcal{O}$. For each $g \in \mathcal{O}$, let $\text{Env}(g) = \{ h \in \mathcal{O} : h \star g \text{ is defined} \}$. The pair $(\mathcal{O}, \star)$ is called a gluing system if it satisfies the following properties for every $g \in \mathcal{O}$:

(GS1) Let $h \in \text{Env}(g)$. Then, $\text{Env}(h \star g) = \text{Env}(h)$. In addition, if $h \in \text{Env}(f)$ or $g \in \text{Env}(f)$, then $h \star g \in \text{Env}(f)$.

(GS2) If $h \in \text{Env}(g)$ and $h \in \text{Env}(f)$, then $(h \star g) \star f = (h \star f) \star g$.

(GS3) If $h \in \text{Env}(g)$ and $g \in \text{Env}(f)$, then $h \star (g \star f) = (h \star g) \star f$.

(GS4) There exists $u \in \text{Env}(g)$ such that $u \star k = k$ for any $k$ for which $\text{Env}(k) = \text{Env}(g)$.

Properties GS2 and GS3 are well-defined given GS1. By Lemma 2.2.1, graphs and the graph gluing operation form a gluing system. In a manner analogous to the way that hierarchical graphs are built upon graphs, we define hierarchical objects in terms of gluing systems.

Definition 3.2.2 Let $(\mathcal{O}, \star)$ be a gluing system. A hierarchical object is a finite list of pairs $\Delta = ((g_1, N_1), \ldots, (g_n, N_n))$ where for each $i$, $g_i \in \mathcal{O}$ and $N_i$ is a list, possibly with repetitions, of integers in the set $\{1, 2, \ldots, i - 1\}$. For each $1 \leq i \leq n$, $\Delta_i = ((g_1, N_1), \ldots, (g_i, N_i))$ is a hierarchical object describing $X(\Delta_i) \in \mathcal{O}$, which is constructed recursively as follows:

\[
\begin{align*}
X(\Delta_i) &:= g_i; \\
&\text{for each } j \text{ in } N_i \text{ do} \\
&\quad \text{compute } X(\Delta_j) \text{ recursively;} \\
&\quad X(\Delta_i) := X(\Delta_i) \star X(\Delta_j) \\
&\text{end}
\end{align*}
\]
We shall now define the notion of a problem on objects belonging to gluing systems. While the technique may seem somewhat artificial, we shall see in the next section and the chapters to come that many real graph problems can be expressed in this manner.

Let \( m \geq 0 \), and for each \( 1 \leq i \leq m \), let \( A_i = (D_i, \cdot_i, \lambda_i) \) be an algebraic structure with the following properties:

- \( \cdot_i \) is a commutative and associative binary operation on the set \( D_i \), and
- \( \lambda_i \in D_i \) is an identity element: for each \( d \in D_i \), \( d \cdot_i \lambda_i = \lambda_i \cdot_i d = d \).

Examples of algebras with these properties are \((\mathbb{N}, +, 0)\), the natural numbers under addition, and \((\mathcal{P}(S), \cup, \emptyset)\), the subsets of a set \( S \) under union.

Let \((\mathcal{O}, *)\) be a gluing system, and let \( T \) be the set of all \((m+1)\)-tuples \( O \times D_1 \times D_2 \times \cdots \times D_m \). The components of each \( T \in T \) are denoted \( T.g, T.d_1, T.d_2, \ldots, T.d_m \), etc.

For \( T \in T \), let \( \text{Env}(T) = \{ S \in T : S.g \in \text{Env}(T.g) \} \). A tuple composition operation \( \circ \) is defined in the obvious way:

\[
T \circ S = (T.g \cdot S.g, T.d_1 \cdot S.d_1, \ldots, T.d_m \cdot S.d_m).
\]

Given the properties of the algebras \( A_1, \ldots, A_m \), it follows that the pair \((T, \circ)\) is a gluing system, and therefore, satisfies properties GS1–GS4.

Let \( S \) be an arbitrary set, and \( \Phi : T \rightarrow \mathcal{P}(S) \) any function such that for each \( T \in T \), \( \Phi(T) \neq \emptyset \). We shall see that it is crucial that \( \Phi \) map each tuple to a non-empty set. \( S \) is the solution space and \( \Phi \) the solution mapping of a problem \( II \) defined on any \( g \in \mathcal{O} \) as follows:

\[
\Pi(g) = \text{any element of } \Phi((g, \lambda_1, \lambda_2, \ldots, \lambda_m)).
\]
In the above definition, \( \Pi \) depends on \( S \) and \( \Phi \). However, when we apply the GBU-method to real problems, \( S \) and \( \Phi \) will depend on the problem \( \Pi \) we wish to solve.

3.2.2 Algorithm GBU

We shall now present algorithm GBU, a hierarchical algorithm whose purpose is to compute \( \Pi(\Delta(A)) \), where \( \Delta \) is any hierarchical object over gluing system \( (\mathcal{O}, \ast) \).

The correctness of algorithm GBU relies upon a generalization of replaceability defined on \( T \) as follows:

**Definition 3.2.3** Let \( S, T \in T \). We say \( S \) is similar to \( T \) with respect to \( \Pi \), denoted \( S \sim_{\Pi} T \), if and only if \( \text{Env}(S) = \text{Env}(T) \) and, for any \( R \in \text{Env}(S) \), \( \Phi(R \circ S) \subseteq \Phi(R \circ T) \).

Similarity is reflexive and transitive, but not necessarily symmetric, so it need not be an equivalence relation. However, as we shall see, symmetry is not crucial to the correctness of algorithm GBU. Note that similarity is an equivalence relation if \( |\Phi(S)| = |\Phi(T)| \) for each \( S, T \in T \).

Let \( B : T \rightarrow T \) be any function such that \( B(R) \sim_{\Pi} R \) for each \( R \in T \). \( B \) is the analog of the burner function used by algorithm BU.

Let \( \Delta = ((g_1, N_1), \ldots, (g_n, N_n)) \) be a hierarchical object over gluing system \( (\mathcal{O}, \ast) \). Applied to \( \Delta \), algorithm GBU uses \( B \) to create sequences \( \bar{T}_1, \ldots, \bar{T}_n \) and \( T^b_1, \ldots, T^b_n \) of tuples in \( T \). We shall prove that for each \( i, T^b_i \sim_{\Pi} \bar{T}_i \sim_{\Pi} T_i \), where \( T_i = (X(\Delta_i), \lambda_1, \ldots, \lambda_m) \). By property GS4, this implies that \( \Phi(T^b_n) \subseteq \Phi(T_n) \), and hence, that algorithm GBU correctly computes \( \Pi(\Delta(A)) \) (recall that \( \Phi(T^b_n) \neq \emptyset \)). Algorithm GBU is given in Figure 3.5.
Algorithm GBU

input: $\Delta = ((g_1, N_1), \ldots, (g_n, N_n))$
output: $\Pi(X(\Delta))$

for $i := 1$ to $n$ do
  $\tilde{T}_i := (g_i, \lambda_1, \lambda_2, \ldots, \lambda_m)$;
  for each $j$ in $N_j$ do
    $\tilde{T}_i := \tilde{T}_i \circ T_j^b$
  end;
  $T_i^b := B(\tilde{T}_i)$
end;

return any element of $\Phi(T_n^b)$

Figure 3.5: Algorithm GBU

Each of the following results has a counterpart in the correctness proof of algorithm BU (see Section 3.1), and the proof techniques employed are virtually identical.

Lemma 3.2.1 If $S \sim_{\Pi} T$, then $R \circ S \sim_{\Pi} R \circ T$ for any $R \in \text{Env}(S)$.

Proof (contrapositive): Suppose $R \circ S$ is not similar to $R \circ T$. By property GS2, $\text{Env}(R) = \text{Env}(R \circ S) = \text{Env}(R \circ T)$. Then, by definition, there exists some $Q \in \text{Env}(R)$ such that $\Phi(Q \circ (R \circ S)) \not\subseteq \Phi(Q \circ (R \circ T))$. But, by GS2, $Q \circ (R \circ S) = (Q \circ R) \circ S$ and $Q \circ (R \circ T) = (Q \circ R) \circ T$. Thus, $\Phi((Q \circ R) \circ S) \not\subseteq \Phi((Q \circ R) \circ T)$. Therefore, $S$ is not similar to $T$. □

Lemma 3.2.2 For $1 \leq i \leq r$, assume $S_i \sim_{\Pi} T_i$ and $R \in \text{Env}(S_i)$. Then, $R \circ S_1 \circ S_2 \circ \cdots \circ S_r \sim_{\Pi} R \circ T_1 \circ T_2 \circ \cdots \circ T_r$.

Proof: Consider any $1 \leq i \leq r$. We have the following:

$R \circ S_1 \circ \cdots \circ S_n = R \circ S_1 \circ \cdots \circ S_{i-1} \circ S_{i+1} \circ \cdots \circ S_n \circ S_i$ by property GS3
\[ \sim_\Pi \quad R \circ S_1 \circ \cdots \circ S_{i-1} \circ S_i+1 \circ \cdots \circ S_n \circ T_i \quad \text{by Lemma 3.2.1} \]
\[ = \quad R \circ S_1 \circ \cdots \circ S_{i-1} \circ T_i \circ S_{i+1} \circ \cdots \circ S_n \quad \text{by property GS3} \]

The lemma follows by the transitivity of similarity and an application of the above transformation for each \( i \). \( \square \)

**Lemma 3.2.3** Let \( \Delta = ((g_1, N_1), \ldots, (g_n, N_n)) \) be a hierarchical object, and for each \( i \), let \( T_i = (X(\Delta_i), \lambda_1, \ldots, \lambda_m) \). Then, \( T_i^b \sim_\Pi T_i \).

**Proof** (induction on \( i \)):

\( i = 1 \): Since \( N_1 \) is an empty list, \( T_1 = T_1 \), and thus, by reflexivity, \( T_1 \sim_\Pi T_1 \). Since \( T_1^b = B(T_1) \), we have that \( T_1^b \sim_\Pi T_1 \) by the assumption that \( B(T) \sim_\Pi R \) for each tuple \( R \).

\( i > 1 \): Let \( N_i = [j_1, j_2, \ldots, j_r] \). Then, \( T_i = (g_i, \lambda_1, \ldots, \lambda_m) \circ T_{j_1}^b \circ \cdots \circ T_{j_r}^b \) and, by definition of \( X(\Delta_i) \), \( T_i = (g_i, \lambda_1, \ldots, \lambda_m) \circ T_{j_1} \circ \cdots \circ T_{j_r} \). By the induction hypothesis, \( T_{j_1}^b \sim_\Pi T_{j_1} \sim_\Pi T_{j_2} \) for each \( j < i \). Thus, by transitivity, \( T_i^b \sim_\Pi T_i \). Then, by Lemma 3.2.2, \( T_i \sim_\Pi T_i \). Since \( T_i^b = B(T_i) \), the lemma follows by the assumption on \( B \). \( \square \)

We now prove that algorithm GBU solves problem \( \Pi \).

**Theorem 3.2.1** Applied to any hierarchical object \( \Delta = ((g_1, N_1), \ldots, (g_n, N_n)) \), algorithm GBU returns \( \Pi(X(\Delta)) \).

**Proof:** Since \( \Phi(R) \neq \emptyset \) for each \( R \in T \), we need only show that \( \Phi(T_n^b) \subseteq \Phi(T_n) \), where \( T_n = (X(\Delta), \lambda_1, \lambda_2, \ldots, \lambda_m) \). By Lemma 3.2.3 and transitivity, \( T_n^b \sim_\Pi T_n \). Thus, \( \text{Env}(T_n^b) = \text{Env}(T_n) \) and, for any \( R \in \text{Env}(T_n) \), \( \Phi(R \circ T_n^b) \subseteq \Phi(R \circ T_n) \). By GS4, there exists some \( R \in \text{Env}(T_n) \) such that \( R \circ T_n = T_n \) and \( R \circ T_n^b = T_n \). Thus, \( \Phi(T_n^b) \subseteq \Phi(T_n) \). \( \square \)
3.2.3 Applying algorithm GBU

Consider the decision problem $\Pi(G) = \text{"is } G \text{ bipartite?"}$ from Section 3.1. We show that the GBU-method can be applied to $\Pi$:

Let $T = G$, the set of all undirected graphs, and let $S = \{\text{true}, \text{false}\}$. For each $G \in T$, let $\Phi(G) = \{\Pi(G)\}$. Clearly, similarity and replaceability are equivalent. By Lemma 3.1.1 and Theorem 3.2.1, algorithm GBU returns an element of $\Phi(X(\Gamma)) = \{\Pi(X(\Gamma))\}$. Therefore, algorithm GBU returns $\Pi(X(\Gamma))$.

Thus, algorithm BU is a special case of algorithm GBU. However, because of its simplicity, we shall apply the BU-method to problems such as bipartiteness testing. We encounter two such problems in Chapter 5.

In the remainder of this section, we use the GBU-method to prove the correctness of the connectivity augmentation algorithm (ConAug) from Section 3.1.4.

Let $G$ be the set of all graphs and let $(N, +, 0)$ be the natural numbers under addition. Then, $T = G \times N$ is the set of all pairs manipulated by algorithm ConAug. Given Lemma 2.2.1 and the properties of addition, it follows that $(T, o)$ is a gluing system. Let the solution space $S$ be the natural numbers $N$, and for any $R \in T$, let $\Phi(R) = \{\text{cn}(R.G) + R.m - 1\}$. Thus, $\Phi(R)$ is a singleton set, so similarity, which we denote $\sim_{cn}$, is an equivalence relation. Therefore, algorithm ConAug is simply an instance of algorithm GBU.

We first prove that the burner preserves similarity.

**Lemma 3.2.4** Let $S = (G, m)$ and $B_{cn}(S) = (G^b, m + q) = S^b$. Then, $S^b \sim_{cn} S$.

**Proof:** Let $(R = (H, p), L) \in \text{Env}(S)$. $G$ and $G^b$ have the same pins, so $\text{Env}(S^b) = \text{Env}(S)$. The burner processes one connected component at a time. Let $C$ be any
connected component of $G$, and let $S' = (G', m')$ be the tuple after the burner processes $C$. We show $\Phi(R \circ L S') = \Phi(R \circ L S)$. Then, $S' \sim_{\text{cn}} S$ since $R$ is arbitrary. The lemma follows by the transitivity of similarity.

**Case 1:** $C$ contains no pins. Then, $G' = G - V(C)$, and $m' = m + 1$. Since $C$ contains no pins, it is also a connected component of $H \circ L G$ containing no pins. Thus, $H \circ L G' = (H \circ L G) - V(C)$, and hence, $\text{cn}(H \circ L G) + p + m = \text{cn}(H \circ L G') + p + (m + 1)$. Thus, $\Phi(R \circ L S') = \Phi(R \circ L S)$.

**Case 2:** $C$ contains pins. Then, $m' = m$, and $G'$ is obtained by exchanging $C^b$ for $C$, where $C^b$ is a path containing the pins in $C$. Clearly $H \circ L G$ and $H \circ L G'$ have the same number of connected components. Then, $\Phi(R \circ L S') = \Phi(R \circ L S)$ since $m' = m$. □

We can now prove the correctness of algorithm ConAug.

**Theorem 3.2.2** Applied to hierarchical graph $\Gamma = (G_1, \ldots, G_n)$, algorithm ConAug returns $\text{cn}(X(\Gamma)) - 1$, the number of edges necessary and sufficient to connect $X(\Gamma)$.

**Proof:** ConAug returns $\text{cn}(T^b_{H1}.G) + T^b_{H1}.m - 1$, the single element of $\Phi(T^b_{H1})$. Then, by Lemma 3.2.4 and Theorem 3.2.1, ConAug returns an element of $\Phi((X(\Gamma), 0))$. By definition, $\Phi((X(\Gamma), 0)) = \{\text{cn}(X(\Gamma)) - 1\}$. □

The GBU-method provides a mechanism for treating many hierarchical algorithms in a uniform manner. Suppose we wish to use the GBU-method to construct and prove the correctness of a hierarchical algorithm for some problem $\Pi$. To apply the GBU-method, we must provide

- a set of tuples $T$ and some composition operation $\circ$ on $T$,
- a solution space $S$,
• a solution mapping \( \Phi : T \rightarrow (\mathcal{P}(S) - \{\emptyset\}) \), and

• a burner function \( B \).

Algorithm GBU consists of a burner and a framework given in Figure 3.5 that invokes the burner. This framework is essentially identical for any instance of algorithm GBU, and need not be repeated. Applied to hierarchical graph \( \Gamma = (G_1, \ldots, G_n) \), algorithm GBU returns \( \Pi(X(\Gamma)) \) provided:

1. \( (T, o) \) is a gluing system,

2. \( B \) preserves similarity, and

3. each element of \( \Phi(T_n) \) is a correct answer to \( \Pi(X(\Gamma)) \).

In Chapters 4 and 6, we shall use this approach on several different problems. We apply the BU-method to the problems considered in chapter 5.

### 3.3 The Complexity of Arithmetic

We analyzed the running time of algorithm ConAug using the unit cost model [1], which assigns one unit of cost to each addition. Another, perhaps more accurate model, is the logarithmic cost model [1]. The logarithmic cost model computes the cost of each arithmetic operation in terms of the number of bits in the operands. For example, adding or comparing two \( n \)-bit numbers takes \( O(n) \) time. Like ConAug, the algorithms presented in the following chapters add or compare numbers that depend on the size of \( X(\Gamma) \). We justify, to some extent, our choice of the unit cost model by proving that the cost of the arithmetic performed by ConAug, its arithmetic cost, is \( O(|\Gamma|^2) \). Thus, ConAug operates in polynomial time under either cost model.
This same arithmetic cost bound holds for the algorithms we develop in the coming chapters.

Let \( \Gamma = (G_1, \ldots, G_n) \). Algorithm ConAug counts the number of connected components of \( X(\Gamma) \). Thus, by Theorem 2.2.1, the largest integer computed by ConAug is at most \( 2|\Gamma| \), and hence, requires at most \( |\Gamma| + 1 \) bits. If we assume every integer manipulated by ConAug is stored in \( |\Gamma| + 1 \) bits, no overflow occurs. For each \( i \), ConAug constructs pairs \( \bar{T}_i \) and \( T_i^b \), each consisting of a graph and an integer. \( G_i \) has at most \(|G_i|\) nonterminals. Thus, \( T_i.m \) is the result of at most \(|G_i|\) additions, each requiring \( O(|\Gamma|) \) time. \( T_i^b \) is formed by burning \( \bar{T}_i \). Each time a connected component containing no pins is found, \( T_i.m \) is incremented. Since \( \bar{T}_i.G \) has size \( O(|G_i|) \), the number of increments is \( O(|G_i|) \). Thus, the arithmetic cost of the \( i \)th iteration of ConAug is \( O(|G_i| \cdot |\Gamma|) \), and therefore, the cost overall is \( O(|\Gamma|^2) \).
4. CONNECTIVITY AUGMENTATION PROBLEMS

Let $C$ be some vertex or edge-connectivity property of graphs. A $C$-augmenting set of $G$ is a set of edges $E_0 \subseteq V(G) \times V(G) - E(G)$ such that $G + E_0$ satisfies $C$. The $C$-augmentation problem is, given a graph $G$, determine the cardinality of a minimum-cardinality $C$-augmenting set of $G$. The three properties of interest to us are bridge-connectivity, biconnectivity, and strong-connectivity. Eswaran and Tarjan [8] have characterized solutions to these three problems on non-hierarchical graphs in terms of quantities easily derived from, respectively, bridge-connectivity forests, biconnectivity forests, and strong-connectivity dags.

Using the results in [8] and the GBU-method, we develop $O(|\Gamma|)$ hierarchical algorithms for bridge-connectivity and biconnectivity augmentation, and a $O(|\Gamma|^3)$ hierarchical algorithm for strong-connectivity augmentation. These results generalize some of the work described in [24], where polynomial-time algorithms are given for biconnectivity and strong-connectivity testing. We also show that weighted versions of the problems studied here are PSPACE-hard.

The organization of this chapter is as follows. In Section 1, we discuss the complexity of augmentation problems. In Sections 2, 3, and 4 we provide, respectively, algorithms for bridge-connectivity, biconnectivity, and strong-connectivity augmentation. We summarize our results and discuss possible extensions in Section 5.
4.1 The Complexity of Augmentation Problems

A natural generalization of the $C$-augmentation problem is, given a graph $G$ having a weight is associated with each pair of non-adjacent vertices, determine the weight of a minimum-weight $C$-augmenting set of $G$. Weighted versions of biconnectivity, bridge-connectivity, and strong-connectivity augmentation have been shown to be NP-hard for non-hierarchical graphs [8]. In this section, we define and analyze weighted versions of connectivity augmentation problems on hierarchical graphs.

Clearly, we must be careful to define the weight function in a succinct way. We consider one technique for doing so. Let $\Gamma = (G_1, G_2, \ldots, G_n)$ be a hierarchical graph. Let $f(u, v)$ be a function that assigns an integer value to every pair $u, v$, where $u \in V(G_i)$ and $v \in V(G_j)$ for some $G_i, G_j \in \Gamma$. For each vertex $v$ in $X(\Gamma)$, denote by $h(v)$ the set of all terminals and pins in $\Gamma$ that are identified to obtain $v$. The weight of a set of edges $E_0$ whose elements are not in $E(X(\Gamma))$ is given by

$$w(E_0) = \sum_{(u,v) \in E_0} g(u,v),$$

where $g(u,v)$ is the minimum of $f(u',v')$, taken over all $u' \in h(u)$ and $v' \in h(v)$.

The weighted $C$-augmentation problem for $\Gamma$ is to compute the weight of minimum-weight $C$-augmenting set of $X(\Gamma)$. The associated decision problem is to determine whether $X(\Gamma)$ has a $C$-augmenting set of weight $B$ or less, for some given integer $B$. Note that the weighted augmentation problem reduces to the unweighted problem if $f(u,v) = 1$ for all pairs on which $f$ is defined.

**Theorem 4.1.1** Weighted strong-connectivity augmentation is PSPACE-hard.

**Proof:** We use a reduction from directed Hamiltonian Circuit (DHC), which is known to be PSPACE-complete for hierarchical graphs [21].
Given an instance $\Gamma = (G_1, \ldots, G_n)$ of DHC, construct an instance $f, B, \Gamma' = (G'_1, \ldots, G'_n)$ of strong-connectivity augmentation: Let $B = |V(X(\Gamma))|$. $B$ can be computed in polynomial time. For $i = 1, \ldots, n$, $V(G'_i) = V(G_i)$ and $E(G'_i)$ consists only of those edges in $G_i$ that are incident on nonterminals. The labels on the edges in $G'_i$ are the same as the labels on the corresponding edges of $G_i$. Note that $V(X(\Gamma')) = V(X(\Gamma))$, but $X(\Gamma')$ has no edges. Define $f$ by $f(u, v) = 1$ if $u, v \in G_i$ and $(u, v) \in E(G_i)$, and $f(u, v) = 2$ otherwise.

By the construction above, $X(\Gamma')$ can be strongly connected by a set of edges of weight $B$ or less if and only if $X(\Gamma)$ contains a directed Hamiltonian circuit.

**Theorem 4.1.2** Weighted bridge-connectivity augmentation is PSPACE-hard.

**Proof:** Identical to that of Theorem 3.1, except that we use a reduction from undirected Hamiltonian circuit (UHC). We limit ourselves to showing that UHC is PSPACE-complete for hierarchical graphs. The proof is by reduction from DHC.

Let $\Gamma = (G_1, \ldots, G_n)$ be an input to DHC. $\Gamma$ is transformed into an instance $\Gamma' = (G'_1, \ldots, G'_n)$ of UHC using an extension of the method employed in [17]. For each pin and terminal vertex $v$ of $G_i$ there are three vertices, $v_1$, $v_2$, and $v_3$, and two edges, $(v_1, v_2)$ and $(v_2, v_3)$, in $G'_i$. For each edge $(u, v)$ in $G_i$, where $u$ and $v$ are pins or terminals, there is an edge $(u_3, v_1)$ in $G'_i$. For each nonterminal $w$ of type $G_j$ in $G_i$, there is a nonterminal $w'$ of type $G'_j$ in $G'_i$. For each edge $(w, v)$ in $G_i$ with label $a$, there are three edges $e_i = (w', v_i)$, $i = 1, 2, 3$, in $G'_i$, where $e_i$ has label $a_i$.

It can be verified that $X(\Gamma')$ has a directed Hamiltonian circuit if and only if $X(\Gamma)$ has an undirected one.

The proof of the next theorem is identical to that of the preceding result.

**Theorem 4.1.3** Weighted biconnectivity augmentation is PSPACE-hard.
We have been unable to show that any of the above problems is in PSPACE.

4.2 Bridge-Connectivity Augmentation

For an undirected graph \( G \), let \( \text{br}(G) \) be the minimum number of edges that must be added to \( G \) to make it bridge-connected. In this section, we present an algorithm BRIDGE that, given a hierarchical graph \( \Gamma \), determines \( \text{br}(X(\Gamma)) \) in \( O(|\Gamma|) \) time.

The key to solving the bridge-connectivity augmentation problem is the following theorem due to Eswaran and Tarjan [8]. Recall the definition of a br-forest given in Definition 2.1.2.

**Theorem 4.2.1** Let \( G \) be an undirected graph, and assume its br-forest \( \text{brf}(G) \) has \( p \) leaves and \( q \) isolated vertices. Then,

\[
\text{br}(G) = \begin{cases} 
0 & \text{if } p + q \leq 1 \\
\left\lfloor \frac{p}{2} \right\rfloor + q & \text{otherwise}
\end{cases}
\]

Thus, \( \text{br}(X(\Gamma)) \) can be computed from the number of leaves and the number of isolated vertices in \( \text{brf}(X(\Gamma)) \).

Let \( G_i \) be a cell with nonterminals \( v_1, \ldots, v_r \), \( H_i = G_i - \{v_1, \ldots, v_r\} \), and \( F_i = \text{brf}(H_i) \). Instead of working with the cells of \( \Gamma \), we manipulate only their br-forests. To do so, we require certain knowledge about the graph \( H_i \):

- For each \( w \in V(H_i) \), \( \alpha(w) \) is the vertex of \( F_i \) such that \( w \in V(\beta(\alpha(w))) \).
- For each \( v \in V(F_i) \), \( \text{pins}(v) \) is the set of pins of \( G_i \) that belong to \( \beta(v) \). We call a vertex \( v \) of \( F \) **good** if \( \text{pins}(v) \) is empty, and **bad** otherwise.

Throughout the algorithm, certain vertices of \( F \) will be deleted. For \( v \in V(F) \), \( \text{mk}(v) \) (the **mark** of \( v \)) indicates the number of neighbors of \( v \) that have been removed.
4.2.1 Composing bridge-connectivity forests

Let \( G_1 \) and \( G_2 \) be graphs, and assume \( G_2 \) has \( r \geq 0 \) pins and \( (G_1, L) \in \text{Env}(G_2) \). Let \( F_1 \) (\( F_2 \)) be a br-forest of \( G_1 \) (\( G_2 \)) having vertex marks. We provide an operation which constructs \( F = \text{brf}(G_1 \circ_L G_2) \) (with marks on its vertices) by gluing \( F_2 \) onto \( F_1 \) in a way that reflects the underlying graph composition. \( F = F_1 \circ_L F_2 \) is a br-forest obtained in two steps. First, \( F_1 \) and \( F_2 \) are glued through a series of vertex identifications to obtain a graph \( H = F_1 \circ_L F_2 \). \( H \) may contain cycles (see Figure 4.2), so it is not necessarily a br-forest. Then, \( F \) is obtained by condensing the bridge-connected components of \( H \). The operation in described in Figure 4.1.

Figure 4.2 illustrates the composition of two br-forests in which the gluing list used in the underlying graph composition is \( L = [(a, 1), (b, 2), (c, 3)] \). Integers labeling vertices of the br-forests indicate the pins that belong to the corresponding bridge-connected components. Vertex marks are not shown.

The next lemma establishes a connection between the composition of br-forests and the composition of their underlying graphs.

**Lemma 4.2.1** Let \( F_1 = \text{brf}(G_1) \), \( F_2 = \text{brf}(G_2) \), and assume \( (G_1, L) \in \text{Env}(G_2) \). Then, \( F_1 \circ_L F_2 = \text{brf}(G_1 \circ_L G_2) \).

**Proof:** Every node \( b \) of \( F_1 \circ_L F_2 \) corresponds to a subgraph of \( G_1 \circ_L G_2 \) that is bridge-connected, but not always maximally so. To see this, note that \( b \) is either a bridge-connected component of \( G_1 \) or \( G_2 \), or is the result of identifying two or more nodes from \( F_1 \) and \( F_2 \). In the former case, the corresponding subgraph of \( G_1 \circ_L G_2 \) is obviously bridge-connected. In the latter case, \( b \) corresponds to a subgraph created by identifying vertices of bridge-connected components in \( G_1 \) and \( G_2 \), a process in which no bridges are created.
input: $F_1, F_2, L$
output: $F = F_1 \circ_L F_2$

begin

STEP 1: Glue Forests
Assume $L = [(a_1, b_1), \ldots, (a_r, b_r)];$
for $i := 1$ to $r$ do
Let $v = \alpha(a_i)$ and $w = \alpha(b_i);$ if $v \neq w$ then
Identify $v$ and $w;$
$mk(vw) := mk(v) + mk(w)$
end

STEP 2: Condense Components
Let $H = F_1 \circ_L F_2$ be the graph created by Glue Forests;
$F := H;$
for each bridge-connected component $B$ in $F$ with $|V(B)| > 1$ do
Let $A$ consist of the vertices in $V(F) - V(B)$ that are
adjacent in $F$ to some vertex of $B;$
Delete $V(B)$ from $F;$
Add a new vertex $x$ to $F$ having $mk(x) = \sum_{b \in V(B)} mk(b);$
Add edges $\{(x, a) : a \in A\}$ to $F$
end
return $F$
end

Figure 4.1: How to glue br-forests
Figure 4.2: Gluing br-forests via $L = [(a, 1), (b, 2), (c, 3)]$
Consider any bridge-connected component \( B \) of \( F_1 \ast L F_2 \). None of the edges of \( B \) are bridges in \( G_1 \ast L G_2 \). Furthermore, every bridge in \( F_1 \ast L F_2 \) must be a bridge in \( G_1 \ast L G_2 \), for its removal increases the number of components of the graph. Thus, each bridge-connected component of \( F_1 \ast L F_2 \) corresponds to a maximally bridge-connected component of \( G_1 \ast L G_2 \), and hence \( F_1 \ast L F_2 = \text{brf}(G_1 \ast L G_2) \). □

Algorithm BRIDGE manipulates triples of the form \( (F, p, q) \), where \( F \) is a br-forest with marked vertices, \( p \) is an integer which records the number of good leaves deleted, and \( q \) is an integer recording the number of good isolated vertices deleted. When convenient, we shall use \( R.F, R.p, \) and \( R.q \) to denote, respectively, the first, second, and third components of a triple \( R \). For a triple \( R \), let \( \text{Env}(R) \) be the set of all ordered pairs \((S, L)\) of triples and gluing lists such that \( (H, L) \in \text{Env}(G) \), where \( H \) and \( G \) are the graphs represented by \( S.F \) and \( R.F \), respectively. Suppose \((S, L) \in \text{Env}(R)\). We extend the \( \ast \) operation to triples by defining \( R \ast_L S = T \) where \( T.F = R.F \ast_L S.F, T.p = R.p + S.p, \) and \( T.q = R.q + S.q \).

The next two lemmas provide some basic properties of the \( \ast \) operator. Let \( R, S, \) and \( T \) be any three triples.

**Lemma 4.2.2** If \((R, L) \in \text{Env}(S)\) and \((R, N) \in \text{Env}(T)\), then \((R \ast_L S) \ast_N T = (R \ast_N T) \ast_L S\).

**Proof:** Let \((R \ast_L S) \ast_N T = X \) and \((R \ast_N T) \ast_L S = Y\). Clearly, \( X.p = Y.p \) and \( X.q = Y.q \). We show that \( X.F = Y.F \).

Let \( C(J) \) denote the result of applying Condense Components to a graph \( J \). Consider any three br-forests \( F, G, \) and \( H \), such that \( F \circ G \) and \( F \circ H \) are defined. By definition, \((F \circ G) \circ H = C(C(F \circ G) \circ H)\). We shall first prove that 
\[ C(C(F \circ G) \circ H) = C((F \circ G) \circ H) \]
Let $B_1$ and $B_2$ be any two bridge-connected components of $F \cdot G$. Denote by $b_1$ ($b_2$) the vertex of $C(F \cdot G)$ that $B_1$ ($B_2$) is condensed to. $B_1$ and $B_2$ are in the same bridge-connected component of $(F \cdot G) \cdot H$ if and only if $b_1$ and $b_2$ are in the same bridge-connected component of $C(F \cdot G) \cdot H$. Furthermore, the $\circ$ operator calculates marks by adding the marks of the vertices in each component. Thus, $C(C(F \cdot G) \cdot H) = C((F \cdot G) \cdot H)$.

From the result above, $C(C(R.F \cdot S.F) \cdot T.F) = C((R.F \cdot S.F) \cdot T.F)$ and $C(C(R.F \cdot T.F) \cdot S.F) = C((R.F \cdot T.F) \cdot S.F)$. Since, as can be easily verified, $(R.F \cdot S.F) \cdot T.F = (R.F \cdot T.F) \cdot S.F$, we have $X.F = Y.F$. □

Lemma 4.2.3 If $(R, L) \in \text{Env}(S)$ and $(S, N) \in \text{Env}(T)$, then $(R \circ_L S) \circ_N T = R \circ (S \circ_N T)$.

Proof: Similar to the proof of Lemma 4.2.2. □

4.2.2 The burner

The burner is a function $B$ that takes a triple $R$ and returns a triple $B(R)$. It is defined in terms of three burning operations given below. Let $R$ be any triple.

1. For a good leaf $a$ of $R.F$, $rl(R, a)$ denotes the result of applying the following operation:

   **Remove Leaf:** Let $b$ be the neighbor of $a$ in $R.F$. Delete $a$ and its incident edge from $R.F$, and increment $mk(b)$. If $mk(a) = 0$, increment $R.p$.

2. For a good isolated vertex $a$ of $R.F$, $rv(R, a)$ denotes the result of applying the following operation:
Remove Isolated Vertex: Delete $a$ from $R.F$. If $\text{mk}(a) = 1$, increment $R.p$. If $\text{mk}(a) = 0$, increment $R.q$.

3. Let $p = a_1, a_2, \ldots, a_r$ ($r > 1$) be a path in $R.F$ of good series vertices. We denote by $\text{cp}(R, p)$ the result of applying the following operation:

Compress Long Path: Replace $p$ in $R.F$ with a good vertex $a$ such that $\text{mk}(a) = \sum_{i=1}^{r} \text{mk}(a_i)$.

$B(R)$ is computed in three steps, each of which applies one of the burning operations until no further applications are possible. The first step applies Remove Leaf, the second applies Remove Isolated Vertex, and the third applies Compress Long Path.

Let us investigate some of the properties of the burner. To do so, we introduce a burning function $B^*$ that is identical to the burner $B$ except that it considers all nodes to be good (i.e., $B^*$ ignores pins). $B^*$ burns the br-forest of any triple to an empty graph in two steps: the first step deletes leaves until each tree becomes an isolated vertex, and the second step removes isolated vertices.

Lemma 4.2.4 Let $J$ be a br-forest with marks on its vertices, and assume $J$ has $l$ leaves $v$ with $\text{mk}(v) = 0$, $m$ isolated vertices $v$ with $\text{mk}(v) = 1$, and $n$ isolated vertices $v$ with $\text{mk}(v) = 0$. If $A = B^*((J, 0, 0))$, then $A.F$ is an empty graph, $A.p = l + m$, and $A.q = n$.

Proof: We have already seen that $A.F$ is an empty graph. Consider any node $v$ of $J$. Let $m^l_k$ denote the value of $\text{mk}(v)$ in $J$, and let $m^f_k$ denote the value of $\text{mk}(v)$ just before $v$ is deleted by Remove Leaf or Remove Isolated Vertex. When a leaf is deleted by Remove Leaf, the mark of the neighboring vertex is incremented. Thus,
If \( v \) is deleted by Remove Leaf, and \( m_f^f = m_i^f + \deg_J(v) - 1 \) if \( v \) is deleted by Remove Isolated Vertex. Then, given how Remove Leaf and Remove Isolated Vertex interpret the marks on the vertices they delete, it follows that \( A.p = l + m \) and \( A.q = n \). \( \square \)

**Corollary 4.2.1** If \( J \) is a br-forest in which \( \text{mk}(v) = 0 \) for each \( v \in V(J) \), and \( A = B(J, 0, 0) \), then \( A.F \) is empty, \( A.p \) is the number of leaves in \( J \), and \( A.q \) is the number of isolated vertices in \( J \).

### 4.2.3 Algorithm BRIDGE

Algorithm BRIDGE is an instance of algorithm GBU defined as follows: Let the solution space \( \mathcal{S} \) be the set of natural numbers \( \mathcal{N} \). Let \( \mathcal{F} \) be the set of all br-forests with vertex marks, let \( (\mathcal{N}, +, 0) \) be the natural numbers under addition, and let \( (\mathcal{N}, \text{max}, 0) \) be the natural numbers under the “max” function. \( T = \mathcal{F} \times \mathcal{N} \times \mathcal{N} \) is the set of all triples manipulated by the burner \( B \). Given Lemmas 4.2.2 and 4.2.3 and the properties of addition and the max function, the pair \( (T, \circ) \) is a gluing system. Define \( \Phi(R) \) for any triple \( R = (F, p, q) \) as follows:

\[
\Phi(R) = \begin{cases} 
\{0\} & \text{if } V(F) = \emptyset \land p + q \leq 1 \\
\lfloor p/2 \rfloor + q & \text{if } V(F) = \emptyset \land p + q > 1 \\
\Phi(B^*(R)) & \text{otherwise}
\end{cases}
\]

In order to prove the correctness of algorithm BRIDGE, we must show that the burner preserves similarity. We denote similarity by \( \sim_{\text{br}} \).
Lemma 4.2.5 For any triple \( R, B(R) \sim_{br} R \).

Proof: We show that \( R' \sim_{br} R \), where \( R' \) is obtained from \( R \) by applying one of three burning operations of \( B \). The lemma then follows by the transitivity of similarity. Let \((S, L) \in \text{Env}(R)\), and let \( A = S \circ_L R \) and \( A' = S \circ_L R' \). We show that \( B^*(A') = B^*(A) \). Therefore, \( \Phi(A') = \Phi(A) \), and hence, \( R' \sim_{br} R \). For a triple \( T \) and a vertex \( x \in V(T.F) \), let \( \text{mk}_T(x) \) denote the mark of \( x \) in \( T.F \).

Remove Leaf

Let \( x \) be a good leaf of \( R.F \) with neighbor \( y \), and let \( R' = rl(R, x) \). Then, it follows that \( \text{mk}_{R'}(y) = \text{mk}_R(y) + 1 \), \( R'.q = R.q \), \( R'.p = R.p \) if \( \text{mk}_R(x) > 0 \), and \( R'.p = R.p + 1 \) if \( \text{mk}_R(x) = 0 \).

Since \( x \) is a good leaf of \( R.F \), \( x \) is a good leaf of \( A.F \) with \( \text{mk}_A(x) = \text{mk}_R(x) \). Let \( z \) be the neighbor of \( x \) in \( A.F \). \( A.F \) and \( A'.F \) are identical except that \( x \) is not a neighbor of \( z \) in \( A'.F \), and \( \text{mk}_{A'}(z) = \text{mk}_A(z) + 1 \). Furthermore, \( A'.q = A.q \), \( A'.p = A.p \) if \( \text{mk}_A(x) > 0 \), and \( A'.p = A.p + 1 \) if \( \text{mk}_A(x) = 0 \). Thus, by Lemma 4.2.4, \( B^*(A') = B^*(A) \).

Remove Isolated Vertex

Let \( x \) be a good isolated vertex of \( R.F \), and let \( R' = rv(R, x) \). Clearly, \( x \) is a good isolated vertex of \( A.F \) with \( \text{mk}_A(x) = \text{mk}_R(x) \). By Lemma 4.2.4, \( B^*(A') = B^*(A) \).

Compress Long Path

Let \( p = a_1, \ldots, a_r \) be a path of good series vertices in \( R.F \), and let \( R' = cp(R, p) \). In \( R' \), the path \( p \) has been replaced by a path \( p' \) consisting of a good series node \( a \) with \( \text{mk}_{R'}(a) = \sum_{i=1}^{r} \text{mk}_R(a_i) \). Also, \( R'.p = R.p \), and \( R'.q = R.q \). Thus, \( A.p = A'.p \) and \( A.q = A'.q \).

Since all vertices on \( p \) are good series vertices, \( p \) either exists in \( A.F \), or is condensed into a single vertex of \( A.F \). Clearly, \( p \) is condensed into a single vertex
of $A.F$ if and only if $p'$ is condensed into a single vertex of $A'.F$. Recalling that
\[ \text{mk}_{R'}(a) = \sum_{i=1}^{n} \text{mk}_R(a_i), \]
it follows that if $p$ and $p'$ are both condensed into single
vertices, $A.F = A'.F$. If neither $p$ nor $p'$ is condensed, then by Lemma 4.2.4, they
have no effect on the second or third components of the triples $B^*(A)$ and $B^*(A').$
Thus, $B^*(A') = B^*(A)$. □

Let $\Gamma = (G_1, \ldots, G_n)$ be a hierarchical graph. For $1 \leq i \leq n$, let the triple $T_i$ be
defined as follows: If $G_i$ has no nonterminals, then $T_i = \langle \text{brf}(G_i), 0, 0 \rangle$. Otherwise, let
$G = G_i - \{v_1, \ldots, v_r\}$, where $v_1, \ldots, v_r$ are the nonterminals of $G_i$, and $v_j$ has type
$G_{ij}$. Then, $T_i = \langle \text{brf}(G), 0, 0 \rangle \circ T_{i_1} \circ \cdots \circ T_{i_r}$. This is precisely the definition used
by the GBU-method. Note that pathnames are not required for algorithm BRIDGE,
so the prefix operation is not used. Clearly $T_i.p = T_i.q = 0$, and $\text{mk}(v) = 0$ for each
vertex $v \in V(T_i.F)$. By Lemma 4.2.1, $T_i.F = \text{brf}(X(\Gamma_i))$.

**Theorem 4.2.2** Applied to $\Gamma = (G_1, \ldots, G_n)$, algorithm BRIDGE returns $\text{br}(X(\Gamma))$.

**Proof:** By Lemma 4.2.5, $B$ preserves similarity. Thus, by Theorem 3.2.1, algo­
rithm BRIDGE returns an element of $\Phi(T_n)$, where $T_n = \langle \text{brf}(X(\Gamma)), 0, 0 \rangle$. By
Corollary 4.2.1 and the definition of $\Phi$, $\Phi(T_n) = \Phi(B^*(T_n)) = \{\text{br}(X(\Gamma))\}$. Thus,
BRIDGE returns $\text{br}(X(\Gamma))$. □

### 4.2.4 Run-time analysis

We now show that algorithm BRIDGE has an implementation that runs in
$O(|\Gamma|)$ time.

Consider the br-forest $F^b_j = T^b_j.F$ for some $j$. Every leaf in $F^b_j$ is bad (contains
a pin), and $F^b_j$ has no consecutive good series vertices. Thus, $|F^b_j| = O(p_j)$, where $p_j$
is the number of pins in cell $G_j$. As a result, if follows that for each $i$, $|F^b_i| = O(|G_i|)$,
where $\bar{T}_i = \bar{T}_i \cdot F$. Bridge-connectivity forests can be constructed in linear time [35], and depth-first search [34] can be used as the basis for a linear-time implementation of the burner. Thus, we need only show that $\bar{T}_i$ can be constructed in time linear in the size of $G_i$.

Let $T = (G_i - \{v_1, \ldots, v_r\}, 0, 0)$, where $v_1, \ldots, v_r$ are the nonterminals of $G_i$, and $v_j$ has type $G_{i,j}$. By definition, $\bar{T}_i = T \cdot T_{i_1}^b \cdot \cdots \cdot T_{i_n}^b$. However, we cannot construct $\bar{T}_i$ in this manner because each application of the $\cdot$ operation takes time linear in the size of the two tuples it composes. Instead, we first compute an intermediate result $S_i = T \cdot T_{i_1}^b \cdot \cdots \cdot T_{i_n}^b$ using the Glue Forests operation. The size of $S_i \cdot F$ is $O(|G_i|)$. By the proof of Lemma 4.2.2, $\bar{T}_i$ can be obtained from $S_i$ by the Condense Components operation. These two steps each take time linear in the size of $G_i$. Thus, the $i$th iteration of BRIDGE takes $O(|G_i|)$ time, and therefore, BRIDGE operates in $O(|\Gamma|)$ time.

### 4.3 Biconnectivity Augmentation

For an undirected graph, let $bi(G)$ denote the minimum number of edges that must be added to $G$ to make it biconnected. In this section, we present an algorithm BICON that, given a hierarchical graph $\Gamma$, determines $bi(X(\Gamma))$ in $O(|\Gamma|)$ time.

As for bridge-connectivity augmentation, we rely on a result by Eswaran and Tarjan [8]. The statement of the theorem is adapted from Rosenthal and Goldner [31].

**Theorem 4.3.1** Let $G$ be an undirected graph with bc-forest $F$. Let $d$ be the degree of the maximum degree c-node in $F$, and assume $F$ has $l$ leaves, and $n$ connected components, $q$ of which are also biconnected. Then,
Algorithm BICON manipulates bc-forests instead of graphs in a manner similar to that for algorithm BRIDGE. Let $G_i$ be a cell of $\Gamma$ with nonterminals $v_1, \ldots, v_r$. Let $H_i = G_i - \{v_1, \ldots, v_r\}$ and $F = \operatorname{bcf}(H_i)$. If $v \in V(H_i)$ is a cutpoint, then $\alpha(v)$ is the unique c-node $c$ of $F$ such that $v = \kappa(c)$. If $v$ is not a cutpoint, then $\alpha(v)$ is the unique b-node $b$ of $F$ such that $v \in V(\beta(b))$. In addition to $\alpha$, we retain in $F$ two other pieces of information about $H_i$. For a node $a$ of $F$, pins$(a)$ denotes the set of pins belonging to the cutpoint or block of $H_i$ corresponding to $a$. Vertex $a$ is called good provided pins$(a)$ is empty, and is called bad otherwise. We must also know which b-nodes of $F$ correspond to "trivial" blocks — blocks that consist of isolated vertices of $H_i$. For a b-node $b$ of $F$, triv$(b)$ is true if biconnected component $\beta(b)$ of $H_i$ is trivial, and false otherwise. As in the algorithm BRIDGE, certain vertices of $F$ will be removed by the burner. For a vertex $a \in V(F)$, mk$(a)$ records the number of neighbors of $a$ that have been removed.

4.3.1 Composing biconnectivity forests

Here we provide a method for constructing the bc-forest of a composition of graphs from the bc-forests of the graphs themselves. For $i = 1, 2$, let $G_i$ be a graph, and let $F_i$ be a bc-forest of $G_i$ with vertex marks. Assume $G_2$ contains $r$ pins, and $(G_1, L) \in \operatorname{Env}(G_2)$. $F = F_1 \circ L F_2$ is a bc-forest obtained in two steps. First, a graph $H = F_1 \bullet L F_2$ is formed by the Glue Forests operation given in Figure 4.3. Glue Forests consists of four cases, and we assume that the conditions for the cases are evaluated in order from top to bottom, and the case applied is the first whose
condition is true. $H$ may contain cycles, so it is not necessarily a bc-forest. The
bc-forest $F$ is obtained from $H$ by the Condense Block operation given in Figure 4.4.

Figure 4.5 illustrates the composition of two bc-forests in which the gluing list
used in the underlying graph composition is $L = [(a,1), (b,2), (c,3)]$.

The next lemma provides a correspondence between the graph and bc-forest
composition operations.

**Lemma 4.3.1** Let $F_1 = \text{bcf}(G_1)$, $F_2 = \text{bcf}(G_2)$, and assume $(G_2, L) \in \text{Env}(G_1)$.
Then, $F_1 \circ_L F_2 = \text{bcf}(G_1 \circ_L G_2)$.

**Proof:** Every b-node $b$ in $F_1 \circ_L F_2$ corresponds to a (not necessarily maximally)
biconnected subgraph of $G = G_1 \circ_L G_2$. This is clearly true if $b$ was a b-node in either
$F_1$ or $F_2$, but not both. It also holds if $b$ resulted from identifying two b-nodes, which
occurs in case 1 of Glue Forests if at least one of the b-nodes is “trivial” (an isolated
b-node whose corresponding block is a single vertex). In this case, no cutpoints are
created.

Consider any c-node $c$ of $F_1 \circ_L F_2$. It was either a cutpoint of only one of $G_1$ or
$G_2$, or it is the result of a vertex identification in cases 3–4 of Glue Forests (when a
trivial b-node is identified with a c-node). Thus, if $c$ is a cutpoint of $F_1 \circ_L F_2$, then
$\kappa(c)$ must be a cutpoint of $G$. If $c$ is not a cutpoint of $F_1 \circ_L F_2$, it must be contained
in some biconnected component $B$ of $F_1 \circ_L F_2$, and must be adjacent only to vertices
of $B$. In this case, removal of $c$ does not increase the number of components of $G$
and, thus, $\kappa(c)$ is not a cutpoint of $G$.

From the above discussion, it follows that each biconnected component of $F_1 \circ_L F_2$
that is condensed to a single b-node by Condense Blocks does indeed correspond
to a block of $G$, and that the c-nodes that are left indeed correspond to cutpoints of
**Glue Forests**

**input:** $F_2$ and $(F_1, L) \in \text{Env}(F_2)$

**output:** $H = F_1 \bullet_L F_2$

**begin**

Assume $L = [(a_1, b_1), \ldots, (a_r, b_r)];$

Let $H$ be the disjoint union of $F_1$ and $F_2;$

for $j := 1$ to $r$ do

Let $v = \alpha(a_i)$ and $w = \alpha(b_j);$  

Case 1: $v$ and $w$ are b-nodes

if triv($v$) $\lor$ triv($w$) then

Identify $v$ and $w;$ mk($vw$) := mk($v$) + mk($w$);

triv($vw$) := triv($v$) $\land$ triv($w$)

else

Add new c-node $c$ and edges $(v, c), (w, c);$ 

Case 2: $v$ and $w$ are c-nodes

Identify $v$ and $w;$ mk($vw$) := mk($v$) + mk($w$);

Case 3: $v$ is a b-node and $w$ is a c-node

if triv($v$) then

Identify $v$ and $w$ ($vw$ is a c-node); pins($vw$) := pins($v$)

else

Add edge $(v, w);$ 

Case 4: $v$ is a c-node and $w$ is a b-node

if triv($w$) then

Identify $v$ and $w$ ($vw$ is a c-node); pins($vw$) := pins($v$)

else

Add edge $(v, w)$

end;

return $H$

**end**

---

Figure 4.3: The Glue Forests operation
**Condense Blocks**

input: $H = F_1 \bullet \_ F_2$

output: $F = F_1 \circ \_ F_2$

begin

$F := H;$

for each block $B$ of $F$ containing at least two b-nodes do

Let $A \subseteq V(F) - V(B)$ be those vertices adjacent to a vertex in $B;$

$C := \{c\text{-nodes } c \in V(B) : c \text{ is bad } \lor \text{ mk}(c) > 0 \lor c \text{ is adjacent to a vertex of } A\};$

$D := V(B) - C;$

Create new b-node $b$ with $\text{mk}(b) = \sum_{v \in D} \text{mk}(v)$ and $\text{triv}(b) = \text{false};$

Set $F := F - D,$ and add $b$ and the edges $\{(v, b) : v \in A \cup C\}$ to $F$

end;

return $F$

end

Figure 4.4: The Condense Blocks operation
Figure 4.5: Gluing bc-forests via $L = [(a, 1), (b, 2), (c, 3)]$
Algorithm BRIDGE manipulates 5-tuples of the form \((F, c, b, l, d)\), where \(F\) is a bc-forest whose vertices have marks, and the other components are integers. We use \(c, b,\) and \(l\) to count, respectively, the number of good trees, good isolated b-nodes, and good leaf b-nodes deleted. We use \(d\) to record the degree of the highest degree good c-node deleted. For convenience, we will often refer to the components of a tuple \(R\) by \(R.F, R.c, R.b, R.l,\) and \(R.d\). For a tuple \(R\), let \(\text{Env}(R)\) be the set of all ordered pairs \((S, L)\) such that \((H, L) \in \text{Env}(G)\), where \(H (G)\) is the graph represented by \(S.F (R.F)\). Let \((S, L) \in \text{Env}(R)\). We extend the \(\circ\) operation to operate on tuples by defining \(S \circ_L R = A\) where \(A.F = S.F \circ_L R.F, A.c = S.c + R.c, \) and \(A.b = S.b + R.b, A.l = S.l + R.l,\) and \(A.d = \max(S.d, R.d)\).

The proofs of the next two lemmas are along the lines of those of Lemmas 4.2.2 and 4.2.3, and are omitted.

**Lemma 4.3.2** If \((R, L) \in \text{Env}(S)\) and \((R, N) \in \text{Env}(T)\), then \((R \circ_L S) \circ_N T = (R \circ_N T) \circ_L S\).

**Lemma 4.3.3** If \((R, L) \in \text{Env}(S)\) and \((S, N) \in \text{Env}(T)\), then \((R \circ_L S) \circ_N T = R \circ_L (S \circ_N T)\).

### 4.3.2 The burner

The burner is a function \(B\) that takes a tuple \(R\) and returns a tuple \(B(R)\). It is defined in terms of three burning operations. Let \(R\) be any triple.

1. For a good leaf \(v\) of \(R.F\), \(rl(R, v)\) denotes the result of applying the following operation:
Remove Leaf: Let \( w \) be the neighbor of \( v \) in \( R.F \). Delete \( v \) and its incident edge from \( R.F \) and increment \( mk(v) \). If \( v \) is a b-node and \( mk(v) = 0 \), increment \( R.l \). If \( v \) is a c-node, set \( R.d := \max(R.d, mk(v) + 1) \).

2. For a good isolated vertex \( v \) of \( R.F \), \( rv(R, v) \) denotes the result of applying the following operation:

   Remove Isolated Vertex: Delete \( v \) from \( R.F \) and increment \( R.c \). If \( v \) is a c-node, set \( R.d := \max(R.d, mk(v)) \). If \( v \) is a b-node and \( mk(v) = 0 \), increment \( R.b \). If \( v \) is a b-node and \( mk(v) = 1 \), increment \( R.l \).

3. Let \( p = c_1, b_1, c_2, b_2, \ldots, c_{r-1}, b_{r-1}, c_r (r > 2) \) be a path in \( R.F \) of good series vertices such that each \( c_j \) is a c-node and each \( b_j \) is a b-node. We denote the result of applying the following operation by \( cp(R, p) \):

   Compress Long Path: Assume \( mk(c_i) = \max_{1 \leq j \leq r}(mk(c_j)) \). If \( i \neq 1 \), swap \( mk(c_1) \) and \( mk(c_i) \). Replace \( b_1, c_2, \ldots, b_{r-1} \) in \( R.F \) with a good b-node \( a \) having \( mk(a) = \sum_{j=1}^{r-1} mk(b_j) + \sum_{j=2}^{r-1} \min(1, mk(c_j)) \).

\( B(R) \) is computed in three steps, each of which applies one of the burning operations until no further applications are possible. The first step applies Remove Leaf, the second applies Remove Isolated Vertex, and the third applies Compress Long Path.

To analyze \( B \) we again utilize a modified version \( B^* \) of \( B \) that considers all vertices to be good.
Lemma 4.3.4 Let $J$ be a bc-forest whose vertices have marks. Let $C$ be the set of c-nodes in $J$, and let $s := \max_{v \in C}(mk(v) + \deg_J(v))$. Assume $J$ has $n$ connected components, $p$ isolated b-nodes $v$ with $mk(v) = 0$, $q$ leaf b-nodes $v$ with $mk(v) = 0$, and $r$ isolated b-nodes $v$ with $mk(v) = 1$. If $A = B^*((J, 0, 0, 0, 0))$, then $A.F$ is the empty graph, $A.c = n$, $A.b = p$, $A.l = q + r$, and $A.d = s$.

Proof: For any vertex $v \in V(J)$, let $m^t_v$ be the value of $mk(v)$ in $J$, and $m^f_v$ be the value of $mk(v)$ just before $v$ is deleted (either by Remove Leaf or Remove Isolated Vertex).

The first step of the burner burns each tree in $J$ down to an isolated vertex. The second step removes the remaining nodes, of which there is one for each connected component of $J$. Thus, $A.F$ is empty, and by definition of Remove Isolated Vertex, $A.c = n$.

Consider any node $v$ of $J$. When a leaf is deleted by Remove Leaf, the mark of the neighboring vertex is incremented. Thus, $m^f_v = m^t_v + \deg_J(v) - 1$ if $v$ is deleted by Remove Leaf, and $m^f_v = m^t_v + \deg_J(v)$ if $v$ is deleted by Remove Isolated Vertex. Then, by definition of Remove Leaf and Remove Isolated Vertex, it follows that $A.b = p$, $A.l = q + r$, and $A.d = s$. □

Corollary 4.3.1 If $J$ is a bc-forest in which $mk(v) = 0$ for each $v \in V(J)$, and $A = B^*((J, 0, 0, 0, 0))$, then $A.F$ is empty, $A.c$ is the number of connected components of $J$, $A.b$ is the number of isolated b-nodes of $J$, $A.l$ is the number of leaf b-nodes of $J$, and $A.d$ is the degree of the highest degree c-node in $J$.

4.3.3 Algorithm BICON

Algorithm BICON is an instance of algorithm GBU defined as follows: Let $S = N$, let $F$ be the set of all bc-forests having marks on their vertices, and let
$T = F \times N \times N \times N \times N$. Given Lemmas 4.3.3 and 4.3.2, the pair $(T, \circ)$ is a gluing system. Define $\Phi(R)$ for any triple $R = (F, c, b, l, d)$ as follows:

$$
\Phi(R) = \begin{cases} 
\{0\} & \text{if } V(F) = \emptyset \land b = 1 \land l = 0 \\
\max(d + c - 2, b + \lceil l/2 \rceil) & \text{if } V(F) = \emptyset \land (b \neq 1 \lor l \neq 0) \\
\Phi(B^*(R)) & \text{otherwise}
\end{cases}
$$

Similarity is denoted $\sim_{bi}$. The following proves that the burner preserves similarity.

**Lemma 4.3.5** For any triple $R$, $B(R) \sim_{bi} R$.

**Proof:** We show that $R' \sim_{bi} R$, where $R'$ is the result of applying any one of the three burning operations to $R$. The result then follows by the transitivity of similarity. Let $(S, L) \in \text{Env}(R)$, and let $A = S \circ_L R$ and $A' = S \circ_L R'$. Given the definition of $\Phi$, we need only show $B^*(A') = B^*(A)$. For a tuple $T$ and vertex $x \in V(T)$, let $\text{mk}_T(x)$ be the mark of $x$ in $T.F$.

**Remove Leaf**

Let $x$ be a good leaf of $A.F$ with neighbor $y$, and let $R' = \text{rl}(R, x)$. Since $x$ is a good leaf of $R.F$, it also a good leaf of $A.F$ with $\text{mk}_A(x) = \text{mk}_R(x)$. Let $z$ be the neighbor of $x$ in $A.F$. By construction, $A.F$ and $A'.F$ are identical except that $x$ is not a neighbor of $z$ in $A'.F$, and $\text{mk}_{A'}(x) = \text{mk}_A(x) + 1$. If $x$ is a c-node, then $A'.d = \max(A.d, \text{mk}_A(x) + 1)$ and $A'.l = A.l$. If $x$ is a b-node, then $A'.d = A.d$, $A'.l = A.l + 1$ if $\text{mk}_A(x) = 0$, and $A'.l = A.l$ if $\text{mk}_A(x) > 0$. Thus, by Lemma 4.3.4, $B^*(A') = B^*(A)$.

**Remove Isolated Vertex**

Let $x$ be a good isolated vertex of $R.F$, and let $R' = \text{rv}(R, x)$. Clearly, $x$ is a good
isolated vertex of $A.F$ with $mk_A(x) = mk_R(x)$. By Lemma 4.3.4, $B^*(A') = B^*(A)$.

**Compress Long Path**

Let $p = c_1, b_1, \ldots, c_{r-1}, b_{r-1}, c_r$ be a path in $R.F$ satisfying the conditions of the Compress Long Path operation, and let $R' = cp(R, p)$. Compress Long Path only affects the bc-forest component of a tuple. In $R'$, $p$ has been replaced by a path $p' = c'_1, a, c'_r$. Let $i$ be the index such that $mk(c_i) = \max_{1 \leq j \leq r} mk(c_j)$, and let $J$ be the number of c-nodes on $p$ with non-zero marks that were compressed into the b-node $a$ of $p'$. Then, by construction, $mk(c'_1) = mk(c_i)$, $mk(c'_r) = mk(c_r)$, and $mk(a) = \sum_{j=1}^{r-1} mk(b_j) + J$.

Since all vertices on $p$ are good, either $p$ is a path in $A.F$, or all c-nodes on $p$ with zero marks and all b-nodes on $p$ have been condensed into a single b-node of $A.F$ by Condense Blocks. $p'$ is a path in $A'.F$ if and only if $p$ is a path in $A.F$. We consider the two cases separately.

Suppose $p$ and $p'$ are not condensed. By Lemma 4.3.4, the paths $p$ and $p'$ can only affect the values of $A.d$ and $A'.d$. Then, since $mk(c_i) = mk(c'_1)$ and these two c-nodes have the highest marks of all c-nodes on $p$ and $p'$, it follows that $A.d = A'.d$. By Lemma 4.3.4, $B^*(A') = B^*(A)$.

Suppose $p$ and $p'$ are condensed. Let $v$ ($v'$) be the vertex of $A.F$ ($A'.F$) that $p$ ($p'$) is condensed to. Then, $A.F$ and $A'.F$ are identical except that $mk(v') = mk(v) + J$, and $v$ has $J$ more neighbors than $v$, all of which are good leaf c-nodes. These neighbors are the c-nodes of $p$ with non-zero marks that were compressed into the vertex $a$ of $p'$. Thus, $A' = A$ if $J = 0$. Assume $J > 0$. Each c-node on $p$ ($p'$) with a nonzero mark is a neighboring leaf c-node of $v$ ($v'$). Then, since $mk(c_i) = mk(c'_1)$, and these c-nodes have the highest marks among all c-nodes on $p$ or $p'$, it follows from Lemma 4.3.4 that $B^*(A') = B^*(A)$. □
The proof of correctness and run-time analysis of algorithm BICON are virtually identical to those given for algorithm BRIDGE.

Consider the cd-forest $F^b_j = T^b_j.F$ for some $j$. Every leaf in $F^b_j$ contains a pin, and the maximum length of any path of good, series vertices is 5. Thus, $|F^b_j| = O(p_j)$, and hence, for each $i$, $|\bar{F}_i| = O(|G_i|)$, where $\bar{F}_i = \bar{T}_i.F$. Biconnectivity forests can be constructed in linear time [34], and depth-first search serves as the basis of an implementation of the burner that operates in linear time. We need only show that $T_i$ can be constructed in linear time in the size of $G_i$. Again, $T_i$ must be constructed in two steps. First, an intermediate tuple $S_i$ is constructed using only the Glue Forests operation. The graph $S_i.F$ has size $O(|G_i|)$. Then, $\bar{T}_i$ is obtained from $S_i$ by the Condense Blocks operation. Each step takes $O(|G_i|)$ time. Thus, algorithm BICON operates in $O(|\Gamma|)$ time. We have the following theorem.

**Theorem 4.3.2** Applied to $\Gamma = (G_1, \ldots, G_n)$, algorithm BICON correctly computes $br(X(\Gamma))$ in $O(|\Gamma|)$ time.

**Proof:** By the preceding argument, BICON operates in $O(|\Gamma|)$ time. By Lemma 4.3.5, the burner preserves similarity. Therefore, by Theorem 3.2.1, algorithm BICON returns an element of $\Phi(T_n)$. By Lemma 4.3.1, $T_n = (\text{bcf}(X(\Gamma)), 0, 0, 0, 0, 0)$. Therefore, by Corollary 4.3.1 and the definition of $\Phi$, BICON returns $\text{bi}(X(\Gamma))$. \(\Box\)

**4.4 Strong-Connectivity Augmentation**

For a directed graph $G$, let $st(G)$ denote the minimum number of directed edges that must be added to $G$ to make it strongly-connected. In this section, we describe an algorithm STRONG that computes $st(X(\Gamma))$ in $|\Gamma|^3$ time for any directed hierarchical graph $\Gamma$. 
Theorem 4.4.1 [8] Let $G$ be any directed graph whose st-dag contains $s$ sources, $t$ sinks, and $q$ isolated vertices. Then,

$$\text{st}(G) = \begin{cases} 
0 & \text{if } q = 1 \text{ and } s = t = 0 \\
\max(s, t) + q & \text{otherwise}
\end{cases}$$

Let $D$ be the st-dag of a directed graph $G$. A vertex $a \in V(D)$ is called good provided $\beta(a)$ contains no pins, and is called bad otherwise. Good sources and sinks are deleted throughout the algorithm. As a result, some of their neighboring vertices may appear to be good sources or sinks, even though they are not. To ensure that we do not count a vertex as a good source or sink when it is not, each vertex $v$ will have two boolean flags: notSRC$(v)$ and notSNK$(v)$. If notSRC$(v) = \text{true}$, then there is, or was, a directed path from a good source to $v$, and hence, $v$ cannot be a good source. If notSNK$(v) = \text{true}$, then there is, or was, a directed path from $v$ to a good sink, and hence, $v$ cannot be a good sink.

Given its similarity to algorithms BRIDGE and BICON, we shall present algorithm STRONG in much less detail. All proofs are omitted.

4.4.1 Composing strong-connectivity dags

Here we describe an operation which constructs the st-dag of a composition of graphs from the st-dags of the graphs. Let $G_1$ and $G_2$ be graphs, and let $D_i = \text{std}(G_i)$, $i = 1, 2$. Assume $(G_1, L) \in \text{Env}(G_2)$. Notice that when two strongly-connected components are glued together by identifying vertex pairs, the result is strongly connected. Thus, we can obtain a st-dag $D = D_1 \circ_L D_2$ in much the same manner as the br-forest composition given in Section 4.2. First, we construct $H = D_1 \bullet_L D_2$ by identifying vertices of $D_1$ and $D_2$ according to $L$. Then, $D$ is obtained by computing the st-dag of $H$. The two boolean flags of a vertex $v$
of $H$ or $D$ are determined by computing the logical OR of the corresponding flags over all the vertices that form $v$. It can be shown that by ignoring vertex flags, $D = \text{std}(G_1 \circ_L G_2)$.

### 4.4.2 The burner

Algorithm STRONG and its burner manipulate 4-tuples of the form $(D, s, t, q)$, where $D$ is a st-dag with vertex flags, $s$ records the number of good sources deleted, $t$ records the number of good sinks deleted, and $q$ records the number of good isolated vertices deleted. We refer to the components of a 4-tuple $R$ by $R.D$, $R.s$, $R.t$, and $R.q$. Let $\text{Env}(R)$ be the set of all 4-tuple and gluing list pairs $(S, L)$ such that $(H, L) \in \text{Env}(G)$, where $H (G)$ is the directed graph represented by $S.D (R.D)$. We extend our composition operator to 4-tuples by defining $S \circ_L R = A$, where $A.D = S.D \circ_L R.D$, $A.s = S.s + R.s$, $A.t = S.t + R.t$, and $A.q = S.q + R.q$.

The burner for strong-connectivity augmentation is simpler than those for bridge-connectivity and biconnectivity augmentation. It is a function $B$ that takes a 4-tuple $R$ and returns a 4-tuple $R' = B(R)$, computed in three steps:

1. Set $R'.D := R.D$. For each vertex $v \in V(R'.D)$, set $\text{notSRC}(v)$ to true if there is a directed path to $v$ from a vertex $w$ such that $w$ is a good source or $\text{notSRC}(w) = \text{true}$. Set $\text{notSNK}(v)$ to true if there is a directed path from $v$ to a vertex $w$ such that $w$ is a good sink or $\text{notSNK}(w) = \text{true}$.

2. Let $m$ be the number of vertices $v$ of $R'.D$ such that $\text{notSRC}(v) = \text{false}$ and either (1) $v$ is a good source, or (2) $v$ is a good isolated vertex with $\text{notSNK}(v) = \text{true}$. Let $n$ be the number of vertices $v$ of $R'.D$ such that $\text{notSNK}(v) = \text{false}$ and either (1) $v$ is a good sink, or (2) $v$ is a good isolated vertex with $\text{notSRC}(v) = \text{true}$. Let $r$ be the number of good isolated vertices $v$ of $R'.D$ such
that notSNK(v) = notSRC(v) = false. Set \( R'.s := R.s + m, R'.t := R.t + n, \)
and \( R'.q := R.q + r. \)

3. Let \( H \) be the transitive closure of \( R'.D \). Set \( R'.D := H - \{U\} \), where \( U \) is the set of all good vertices in \( H \).

4.4.3 Algorithm STRONG

The 4-tuples manipulated form a gluing system, and thus, algorithm STRONG in an instance of algorithm GBU. Again, define \( B^* \) to be identical to \( B \) except that it ignores pins. Suppose \( D \) is a st-dag in which notSRC(v) = notSNK(v) = false for every vertex \( v \) of \( D \). If \( S = B^*((D, 0, 0, 0)) \), then \( S.D \) is empty, \( S.s \) is the number of sources of \( D \), \( S.t \) is the number of sinks of \( D \), and \( S.q \) is the number of isolated vertices of \( D \). Thus, for any tuple \( R = (D, s, t, q) \) be any tuple, we define \( \Phi(R) \) as follows:

\[
\Phi(R) = \begin{cases} 
0 & \text{if } V(D) = \emptyset \land q = 1 \land s = t = 0 \\
\{\text{max}(s, t) + q\} & \text{if } V(D) = \emptyset \land (q \neq 1 \lor s > 0 \lor t > 0) \\
\Phi(B^*(R)) & \text{otherwise}
\end{cases}
\]

Let \( R \) be any tuple and \( R' = B(R) \). Then, for any \( v, w \in V(R'.D) \), there is an edge \((v, w) \in E(R'.D)\) if and only if there is a directed path from \( v \) to \( w \) in \( R.D \). A straightforward analysis of how the flags notSRC(x) and notSNK(x) and the fields of \( R' \) are set shows that \( R' \sim_{st} R \). Then, given the definition of \( \Phi \), it follows that algorithm STRONG returns \( st(X(\Gamma)) \) when applied to any directed hierarchical graph \( \Gamma \).

We now provide a brief analysis of the running time of STRONG. Let \( R \) be any
tuple and \( R' = B(R) \). If the graph corresponding to \( R \cdot D \) has \( p \) pins, it follows that \( R' \cdot D \) has \( O(r) \) vertices, but may have \( O(r^2) \) edges. Let \( D_j^b = T_j^b \cdot D \). Because \( D_j^b \) can have \( O(p_j^2) \) edges, where \( p_j \) is the number of pins of \( G_j \), creating the intermediate st-dag \( T_j^i \cdot D \) can take \( O(|V(G_j)|^3) \) time. However, \( |V(T_i^i \cdot D)| = O(|G_i|) \), so the time required to construct \( T_i \cdot D \) dominates the time required to compute its transitive closure. The first and second steps of the burner can be implemented by a breadth-first traversal [2] forward from each source, and then backward from each sink. Hence, algorithm STRONG operates in \( O(|\Gamma|^3) \) time. We have the following theorem.

**Theorem 4.4.2** Applied to directed hierarchical graph \( \Gamma = (G_1, \ldots, G_n) \), algorithm STRONG correctly computes \( st(X(\Gamma)) \) in \( O(|\Gamma|^3) \) time.

### 4.5 Discussion

We have provided hierarchical algorithms for bridge-connectivity, biconnectivity, and strong-connectivity augmentation problems on hierarchical graphs. The bridge-connectivity and biconnectivity algorithms operate in \( O(|\Gamma|) \) time under the unit cost model, and \( O(|\Gamma|^2) \) time under the logarithmic cost model. The strong-connectivity algorithm operates in \( O(|\Gamma|^3) \) time under either cost model.

Our algorithms also solve the associated decision problem of determining whether \( X(\Gamma) \) bridge-connected (biconnected, strongly-connected). Thus, our results generalize those of Lengauer and Wanke [24] at the expense of maintaining quantities other than just the graphs. The techniques we have used appear to have some limitations, the most important of which is that our methods do not yield any obvious way of generating an augmenting set of edges. The generation problem can be solved in polynomial time for non-hierarchical graphs [8,31]. Clearly, an augmenting set for
$X(\Gamma)$ can be of size exponential in $|\Gamma|$, and therefore, any algorithm for generating such sets requires exponential time in the worst case.

We are left with the question of whether it is possible to generate an augmenting set in polynomial space without expanding $\Gamma$. Our techniques and those of [8,31] seem unsuited for this purpose. In contrast, Lengauer [19] gave a method for listing the edges in a minimum spanning tree of $X(\Gamma)$ in $O(m \log \log^* m + n + |X(\Gamma)|)$ time and $O(|\Gamma|)$ space, where $m$ is the number of edges and $n$ the number of vertices in $\Gamma$ (similar problems are considered in Chapter 6). Unlike the MST problem, however, generating a biconnectivity-augmenting set of $X(\Gamma)$, for instance, is not a matter of excluding edges, but of adding them. Adding edges to a hierarchically defined graph can create irregularities which may no longer allow it to be succinctly described. We have not found a way to generate augmenting sets using polynomial space in the size of $\Gamma$, nor proved that it cannot be done.
5. SUBGRAPH HOMEOMORPHISM PROBLEMS

A class of graphs can sometimes be characterized as the set of graphs that contain no subgraph homeomorphic from a member of some finite set of forbidden graphs. For example, a graph is planar [14] if and only if it contains no subgraph homeomorphic from $K_5$ or $K_{3,3}$, the Kuratowski graphs, which are shown in Figure 5.1. Series-parallel [7] graphs are those graphs containing no subgraph homeomorphic from $K_4$, and outer-planar [14] graphs are those containing no subgraph homeomorphic from $K_4$ or $K_{2,3}$, the graphs given in Figure 5.2.

In this chapter, we present linear-time hierarchical algorithms for determining whether or not $X(\Gamma)$ is series-parallel or outer-planar. To some extent, our algorithms are similar to a linear-time hierarchical planarity testing algorithm described by Lengauer [20]. On the other hand, the techniques we use also allow for the generation of a forbidden subgraph of $X(\Gamma)$, if one exists, using work-space linear in the size of $\Gamma$. This is due to the fact that our algorithms rely entirely on edge contraction and deletion.

![Figure 5.1: The Kuratowski graphs](image)

$K_5$  $K_{3,3}$
This chapter is organized as follows. In Section 1, we investigate the affects of edge deletion and contraction on replaceability (Definition 3.1.1) with respect to the property of being series-parallel or outer-planar. In Sections 2 and 3 we present and analyze, respectively, our hierarchical series-parallelness and outer-planarity tests. In Section 4, we describe an algorithm for generating the edges of a forbidden subgraph of the expansion of a hierarchical graph $\Gamma$. We summarize our results in Section 5.

5.1 Replaceability and Edge Deletion and Contraction

For a graph $G$, let $sp(G) = \text{"is } G \text{ series-parallel?"}$, and let $op(G) = \text{"is } G \text{ outer-planar?"}$. In Sections 5.2 and 5.3 we present hierarchical algorithms for these problems. The fundamental operations used by the burners of these algorithms are edge deletion and contraction. In this section, we investigate the effects of edge deletion and contraction on replaceability with respect to problems $sp$ and $op$.

A straightforward connection between edge deletion and replaceability is given in the following theorem.

**Theorem 5.1.1** Let $\mathcal{F}$ be any set of graphs without redundant edges. For any graph $G$, let $\Pi(G)$ be true if and only if $G$ contains no subgraph homeomorphic from a member of $\mathcal{F}$. Then, for any graph $G$ with a redundant edge $e$, $G - e \approx_\Pi G$. In
addition, $G - e$ is biconnected if $G$ is biconnected.

Proof: Clearly, $G - e$ is biconnected if $G$ is biconnected. Consider any $(H, L) \in \text{Env}(G)$. Since $e$ is redundant in $G$, $e$ is redundant in $H \circ L G$. Then, since no member of $\mathcal{F}$ has a redundant edge, $\Pi((H \circ L G) - e) = \Pi(H \circ L G)$. The theorem follows since $e \in E(G)$ implies $(H \circ L G) - e = H \circ L (G - e)$. □

Let $G$ be biconnected, and recall the definition given in Section 2.1.3 of a separation pair of $G$. For $\{u, v\} \subseteq V(G)$, we shall often write $\{u, v\}$ separates $G$ (does not separate $G$) to indicate that $\{u, v\}$ is (is not) a separation pair of $G$. Suppose $G$ is biconnected and separated by $\{u, v\}$. If $G$ is simple, then there exist distinct vertices $x, y \in V(G) - \{u, v\}$ such that every path connecting $x$ and $y$ passes through either $u$ or $v$. Thus, $G - \{u, v\}$ is not connected, and we say that $\{u, v\}$ separates $x$ from $y$. However, if $G$ is not simple, $G - \{u, v\}$ may be connected.

Lemma 5.1.1 Let $G$ be a biconnected graph with a non-redundant edge $e = (u, v)$ such that $\{u, v\}$ does not separate $G$. Then, $G/e$ is biconnected. Furthermore, for any graph $A$, $A \subseteq G/e$ implies $A \subseteq G$.

Proof: Clearly, $A \subseteq G/e$ implies $A \subseteq G$ for any graph $A$. $G/e$ can have no cutpoint other than the vertex $uv$. Since $G$ is biconnected and $e$ is not redundant, $G/e$ has no loops. Let $x, y \in V(G) - \{u, v\}$. Since $\{u, v\}$ does not separate $G$, $x$ and $y$ are connected by a path in $G$ avoiding $u$ and $v$. This path avoids $uv$ in $G/e$. Thus, $G/e$ is biconnected. □

5.1.1 Contraction in series-parallel graphs

Our goal is to describe edges $e$ of a biconnected series-parallel graph $G$ such that $G/e$ is biconnected, series-parallel, and replaceable with $G$. The following result will
prove useful in describing such edges.

Lemma 5.1.2 Let $G$ be the set of all connected graphs with two coupling vertices $u$ and $v$. The following are the $\approx_{SP}$-equivalence classes of $G$:

(SP1) Graphs $G$ such that $K_4 \subseteq G$.

(SP2) Graphs $G$ such that $K_4 \not\subseteq G$, but $K_4 \subseteq G + (u,v)$.

(SP3) Graphs $G$ such that $K_4 \not\subseteq G + (u,v)$.

Proof: It is clear that SP1, SP2, and SP3 partition $G$, and that graphs from different sets of the partition are not replaceable. It remains to show that any two graphs from the same set of the partition are replaceable.

Let $J$ be any graph, and let $L = [(u', u), (v', v)]$, where $u' \neq v'$ are vertices of $J$.

For $G \in SP1$, $J \circ L G$ is never series-parallel. For $G \in SP2$, $K_4 \subseteq J \circ L G$ if and only if $K_4 \subseteq J$ or $u'$ and $v'$ are connected in $J$. For $G \in SP3$, $K_4 \subseteq J \circ L G$ if and only if $K_4 \subseteq J$ or $K_4 \subseteq J + (u', v')$. Thus, SP1, SP2, and SP3 are the $\approx_{SP}$-equivalence classes of $G$. □

Lemma 5.1.3 Let $G$ be a biconnected graph with a non-redundant edge $e = (u,v)$ such that $\{u,v\}$ does not separate $G$. For $1 \leq i \leq m$, let $H_i$ be a connected graph with coupling vertices $x_i$ and $y_i$, and assume that for any $v \in V(H_i)$, $H_i$ has a simple path from $x_i$ to $v$ to $y_i$. Let $L_i = [(a_i, x_i), (b_i, y_i)]$ be a gluing list for gluing $H_i$ onto $G$, and assume $a_i, b_i \notin \{u,v\}$. Then, for each $i$:

1. $F_i = G \circ L_1 H_1 \circ L_2 \cdots \circ L_i H_i$ is biconnected and not separated by $\{u,v\}$,

2. $F'_i = (G/e) \circ L_1 H_1 \circ L_2 \cdots \circ L_i H_i$ is biconnected, and
3. \( \text{sp}(F'_i) = \text{sp}(F'_i) \) if \( G \) is series-parallel.

Proof (induction on \( i \)):

\( i = 1 \): By Lemma 5.1.1, \( G/e \) is biconnected. It follows from the properties of \( H_1 \) that \( F_1 \) and \( F'_1 \) are biconnected. Since \( H_1 \) is connected and \( \{u, v\} \) does not separate \( G \), \( \{u, v\} \) does not separate \( F'_i \). Thus, parts 1 and 2 hold.

Assume \( G \) is series-parallel. By Lemma 5.1.1, \( G/e \) is also series-parallel. We can rewrite \( F_1 \) as \( H_1 \circ N_1 \) \( G \) and \( F'_1 \) as \( H_1 \circ N_1 \) \( G/e \), where \( N_1 = [(x_1, a_1), (y_1, b_1)] \).

Thus, we need only show that \( G \) and \( G/e \) belong to the same \( \approx_{\text{sp}} \)-equivalence classes, assuming their coupling vertices are \( a_1 \) and \( b_1 \). Since \( G \) and \( G/e \) are series-parallel, neither belongs to \( \text{SP}1 \). We show \( G \in \text{SP}2 \) if and only if \( G/e \in \text{SP}2 \).

Suppose \( G \not\in \text{SP}2 \). Then, \( G \in \text{SP}3 \), and hence, \( G + (a_1, b_1) \) is series-parallel. Clearly, \( G/e + (a_1, b_1) \) is also series-parallel. Thus, \( G/e \not\in \text{SP}2 \).

Suppose \( G \in \text{SP}2 \). Then, \( K_4 \subseteq G + (a_1, b_1) \). Then, since \( G \) is biconnected and series-parallel, it must contain a subgraph \( F \) of the form

\[
F: \quad \begin{array}{c}
\bullet & \circ & \bullet \\
| & \hspace{1cm} | & | \\
\hspace{1cm} p & \hspace{1cm} & \\
\bullet & \circ & \bullet \\
\end{array}
\]

where dashed lines represent paths of one or more edges. Suppose the path in \( F \) labeled \( p \) consists only of the edge \( e \). Then, since \( \{u, v\} \) does not separate \( G \), there is a path in \( G \) connecting \( a_1 \) and \( b_1 \) that avoids \( u \) and \( v \). Thus, \( K_4 \subseteq G \), a contradiction. Therefore \( p \neq e \), and hence \( K_4 \subseteq F/e + (a_1, b_1) \). Since \( F/e \subseteq G/e \), \( G/e \in \text{SP}2 \).

Therefore, \( G \) and \( G/e \) belong to the same \( \approx_{\text{sp}} \)-equivalence classes, and hence, \( \text{sp}(F_1) = \text{sp}(F'_i) \).
By the induction hypothesis, $F_{i-1}$ and $F'_{i-1}$ are biconnected and $F_{i-1}$ is not separated by $\{u,v\}$. Then, by the same argument as used in the basis, $F_i = F_{i-1} \circ L_i \cdot H_i$ and $F'_i = F'_{i-1} \circ L_i \cdot H_i$ are biconnected and $F_i$ is not separated by $\{u,v\}$. Thus, parts 1 and 2 hold.

Assume $G$ is series-parallel. Then, by the induction hypothesis, $\text{sp}(F_{i-1}) = \text{sp}(F'_{i-1})$. If $F_{i-1}$ and $F'_{i-1}$ are not series-parallel, then neither are $F_i$ and $F'_i$.

Suppose $F_{i-1}$ and $F'_{i-1}$ are series-parallel. Since $a_i, b_i \notin \{u,v\}$, we can express $F_i$ as $H_i \circ N_i \cdot F_{i-1}$ and $F'_i$ as $H_i \circ N_i \cdot F'_{i-1}$, where $N_i = [(x_i, a_i), (y_i, b_i)]$. By applying the same argument as used in the basis, it follows that $F_{i-1}$ and $F'_{i-1}$ are in the same $\approx_{\text{sp}}$-equivalence class (assuming their coupling vertices are $a_i$ and $b_i$). Thus, $\text{sp}(F_i) = \text{sp}(F'_i)$. □

We now prove the main result concerning contraction and replaceability.

**Theorem 5.1.2** Let $G$ be a biconnected series-parallel graph with $r > 0$ coupling vertices, and let $e = (u,v)$ be a non-redundant edge of $G$ such that $u$ and $v$ are not coupling vertices and $\{u,v\}$ does not separate $G$. Then, for any $(H, L) \in \text{Env}(G)$, $\text{sp}(H \circ L \cdot G) = \text{sp}(H \circ L \cdot (G/e))$.

**Proof:** If $H$ is not series-parallel, then neither are $H \circ G$ and $H \circ (G/e)$. Assume $H$ is series-parallel. Suppose $r = 1$. Then, $G$ is a block of $H \circ G$. By Lemma 5.1.1, $G/e$ is biconnected and series-parallel, and hence, is a block of $H \circ (G/e)$. Since $K_4$ is biconnected, a graph is series-parallel if and only if each of its blocks is series-parallel. Thus, $\text{sp}(H \circ G) = \text{sp}(H \circ (G/e))$.

Assume $r > 1$. $G$ is a subgraph of some block $A$ of $H \circ G$, and every other block of $H \circ G$ is a block of $H$. It can be easily verified that $\{u,v\}$ does not separate $A$. Then, by Lemma 5.1.1, $A/e$ is a block of $H \circ (G/e)$, and hence, $G/e$ is a subgraph of
Thus, since we need only consider the blocks $A$ and $A/e$, we can assume $H \circ G$ is biconnected without loss of generality.

Let $R$ be the set of coupling vertices of $G$. For convenience, we assume that $H$ also contains $R$, and that $H \circ G$ is formed by identifying corresponding vertices. Consider the following tree $T$ whose leaves, $x$, $y$, and $z$, are the only vertices of $R$ in $T$.

![Diagram](image)

We consider two cases, depending on whether or not $H$ contains a subgraph such as $T$.

**Case 1:** $H$ contains $T$.

Since $G$ is biconnected, it contains one of the following two graphs

![Graphs](image)

where, as before, dashed lines denote paths. Then, $K_4 \subseteq H \circ G$. Since $u, v \notin R$, $G/e$ also contains one of these two graphs, and therefore, $K_4 \subseteq H \circ G/e$.

**Case 2:** $H$ contains no such subgraph $T$.

Recall that $H$ is assumed to be series-parallel. We show that $H$ can be decomposed into series-parallel graphs $h_1, \ldots, h_s$ such that for each $1 \leq i \leq s$:

1. (P1) $h_i$ contains two vertices of $R$, say $x_i$ and $y_i$,
2. (P2) $h_i$ and $h_j$ ($i \neq j$) have at most one common vertex, which must be a vertex of $R$.
3. (P3) for each $v \in V(h_i)$, $h_i$ has a simple path from $x_i$ to $v$ to $y_i$. 


Consider any \( x, y \in R \). Suppose \( H \) has a simple path \( p \) that connects \( x \) and \( y \) and contains no vertices of \( R \) except for \( x \) and \( y \). Let \( h_{x,y} \) be the subgraph of \( H \) containing \( x, y \), and all such paths \( p \). These subgraphs clearly satisfy P1 and P3. If \( V(H) = R \), then P2 is also satisfied. Assume \( v \in V(H) \) is not in \( R \). Since \( H \circ G \) is biconnected, there exist \( x, y \in R \) such that \( v \in V(h_{x,y}) \). However, since \( H \) does not contain the tree \( T \), every path from \( v \) to any \( z \in R - \{x, y\} \) must contain either \( x \) or \( y \). Therefore P2 is satisfied.

Let \( h_1, \ldots, h_s \) be the decomposition of \( H \) as specified. We can rewrite \( H \circ G \) as \( G \circ h_1 \circ \cdots \circ h_s \), where each \( h_i \) is glued onto \( G \) by its two unique vertices of \( R \). Then, by Lemma 5.1.3, it follows that \( sp(H \circ G) = sp(H \circ (G/e)) \). \( \Box \)

5.1.2 Contraction in outer-planar graphs

Every outer-planar graph is also series-parallel. Thus, given the results for \( K_4 \), we need only concern ourselves with \( K_{2,3} \).

Lemma 5.1.4 Let \( G \) be the set of all connected graphs \( G \) having two coupling vertices \( u \) and \( v \). Let \( s \) and \( t \) be vertices not in any \( G \in G \). The following are the \( \approx_{op} \)-equivalence classes of \( G \):

(\text{OP1}) Graphs that are not outer-planar.

(\text{OP2}) Outer-planar graphs \( G \) such that \( G + (u,v) \) is not outer-planar.

(\text{OP3}) Graphs \( G \) such that \( G + (u,v) \) is outer-planar, but \( G + \{(u,s),(s,v)\} \)

is not outer-planar.

(\text{OP4}) Graphs \( G \) such that \( G + (u,v) \) and \( G + \{(u,s),(s,v),(u,t),(t,v)\} \) are outer-planar, but \( G + \{(u,s),(s,v),(u,t),(t,v)\} \) is not outer-planar.
(OP5) **Outer-planar graphs** $G$ whose only paths from $u$ to $v$ are single edges.

**Proof:** The proof is similar to that of Lemma 5.1.2. The sets described above partition $G$, and graphs from different sets are not replaceable. We show that two graphs belonging to the same set are replaceable. Let $G \in \mathcal{G}$. Let $H$ be any graph, and let $L = [(u', u), (v', v)]$, where $u' \neq v'$ are vertices of $H$. We assume $H$ is outer-planar, for otherwise $H \circ_L G$ is never outer-planar. Let $J = H \circ_L G$.

If $G \in \text{OP}1$, $J$ is never outer-planar. If $G \in \text{OP}2$, $J$ is not outer-planar if and only if $u'$ and $v'$ are connected in $H$. If $G \in \text{OP}3$, $J$ is not outer-planar if and only if $u'$ and $v'$ are connected in $H$ by a path having two or more edges. If $G \in \text{OP}4$, $J$ is not outer-planar if and only if $H + \{(u', x), (x, v')\}$ is not outer-planar, where $x \notin V(H)$. If $G \in \text{OP}5$, $J$ is not outer-planar if and only if $H + (u', v')$ is not outer-planar. It follows that two graphs belonging to the same set in the partition are replaceable. □

**Theorem 5.1.3** Let $G$ be a biconnected outer-planar graph having $r > 0$ coupling vertices, and let $e = (u, v) \in E(G)$ be a non-redundant edge such that $u$ and $v$ are not coupling vertices and $\{u, v\}$ does not separate $G$. Then, for any $(H, L) \in \text{Env}(G)$, $\text{op}(H \circ_L G) = \text{op}(H \circ_L (G/e))$.

**Proof:** If $H$ is not outer-planar, the result follows. Assume $H$ is outer-planar. By an argument similar to that given in the proof of Theorem 5.1.2, the result holds when $r = 1$. Assume $r > 1$. If $H \circ (G/e)$ is not outer-planar, then neither is $H \circ G$. Suppose $H \circ G$ is not outer-planar, but $H \circ (G/e)$ is outer-planar. We shall arrive at a contradiction. By Theorem 5.1.2, we may assume $K_4 \not\subseteq H \circ G$ and $K_{2,3} \subseteq H \circ G$. Since $K_{2,3}$ is biconnected and $H$ is outer-planar, we may assume, without loss of
generality, that $H \circ G$ is biconnected. By Lemma 5.1.1, $G/e$ is biconnected, and hence, so is $H \circ (G/e)$. Since $e \in E(G)$, $(H \circ G)/e = H \circ (G/e)$.

Consider $J \subseteq H \circ G$ homeomorphic from $K_{2,3}$. $J$ has two degree 3 vertices that we call corners. If either $u$ or $v$ does not belong to $J$, then $J/e = J$, and hence, $K_{2,3} \subseteq (H \circ G)/e$, a contradiction. If $e \notin E(J)$, then $J + e$ contains a subgraph $J'$ such that $K_{2,3} \preceq J'$, and $e \in E(J')$. Thus, we assume $e \in E(J)$. By assumption, $K_{2,3} \not\preceq J/e$. Thus, an endpoint of $e$ (assume $u$) is a corner of $J$. Let $x$ be the other corner of $J$, and let $T$ be the maximal subgraph of $G$ that is also a subgraph of $J$. $T$ and $J$ are shown below.

Note that the path $p$ connecting $v$ and $x$ in $J$ must be a single edge, for otherwise $K_{2,3} \preceq J/e$. By the same reasoning, $x = c$ in $J$. Since $G$ is biconnected and not separated by $\{u, v\}$, one of the following two graphs must be a subgraph of $G$.

Suppose $A \subseteq G$. Even if $d = c$, $K_{2,3} \preceq A$, a contradiction. Suppose $B \subseteq G$. Then, $z = a$ and the path $s$ connecting $a$ and $u$ is a single edge, for otherwise, $K_{2,3} \preceq B$, a contradiction. However, $J$ has a path $q$ that connects $b$ and $c$, and is disjoint from $T$ except for its endpoints. Thus, $K_4 \subseteq H \circ G$, a contradiction. □
5.2 Recognition of Series-Parallel Graphs

In this section, we present a hierarchical algorithm that, given \( \Gamma \), determines \( sp(X(\Gamma)) \) in time linear in the size of \( \Gamma \). By Theorems 3.1.1 and 3.1.2, we need only provide a burner that has a linear-time implementation and produces replaceable graphs whose sizes are linear in their number of pins.

5.2.1 The burner

Let \( G \) be any graph. If \( G \) is not series-parallel, then the burner, which we denote \( B_{sp} \), can preserve replaceability by producing any graph \( G^b \) containing a subgraph homeomorphic from \( K_4 \).

Assume \( G \) is series-parallel. For any block \( A \) of \( G \), define the coupling vertices of \( A \) to be those that are pins or cutpoints of \( G \). Thus, the coupling vertices of \( A \) attach it to the rest of \( G \) and to any graph \( G \) is glued onto. The burner uses three burning operations. Following the description of each operation is a proof that it preserves replaceability. Then, we present the burner and prove that it produces replaceable graphs whose sizes are linear in their number of pins.

**Definition 5.2.1** A block \( A \) of \( G \) is **useless** if \( A \) has no coupling vertices, or its only coupling vertex is a cutpoint of \( G \).

For a useless block \( A \) of \( G \), \( db(G, A) \) denotes the result of applying the following operation:

**Delete Block**: Delete \( V(A) - C \) from \( G \), where \( C \) is the set of coupling vertices of \( A \).
Lemma 5.2.1 Let $G$ be a series-parallel graph, and let $A$ be any useless block of $G$. Then, $\text{db}(G, A) \approx_{\text{sp}} G$.

Proof: Consider any $(H, L) \in \text{Env}(G)$. $G$ and $G' = \text{db}(G, A)$ have the same coupling vertices. The only pin, if any, of $A$ is its single coupling vertex. Thus, $A$ is a useless block of $H \circ_L G$, and hence, $H \circ_L G' = \text{db}(H \circ_L G, A)$. A graph is series-parallel if and only if each of its blocks is series-parallel. Thus, $\text{sp}(H \circ_L G') = \text{sp}(H \circ_L G)$. □

The following operation is motivated by Lemma 5.1.2, which describes the $\approx_{\text{sp}}$-equivalence classes $\text{SP}_1, \text{SP}_2,$ and $\text{SP}_3$ of graphs having two coupling vertices.

Definition 5.2.2 A block-path of $G$ is a sequence $p = A_1, A_2, \ldots, A_m$ of blocks of $G$ such that (1) each $A_i$ has exactly two coupling vertices $c_{i-1}$ and $c_i$, and these vertices are cutpoints of $G$, and (2) $V(A_i) \cap V(A_{i+1}) = \{c_i\}$ for each $i < m$. The block-path $p$ is compressible if $m > 1$ and the only pins among $c_0, c_1, \ldots, c_m$, if any, are $c_0$ and $c_m$.

For a compressible block-path $p = A_1, \ldots, A_m$ of $G$, $\text{cbp}(G, p)$ denotes the result of applying the following operation:

Compress Block-Path: Find an $i$ such that $A_i \in \text{SP}_2$. If no such $i$ exists, let $i = 1$. Find a simple path $q_1$ connecting $c_0$ and $c_{i-1}$, and a simple path $q_2$ connecting $c_i$ and $c_m$. For each $j \neq i$, delete from $G$ all vertices in $A_j$ except those on $q_1$ or $q_2$. Contract the edges of $q_1$ into $c_0$ and those of $q_2$ into $c_m$.

The path $q_1$ contains cutpoints $c_0, c_1, \ldots, c_{i-1}$, and the path $q_2$ contains cutpoints $c_i, c_{i+1}, \ldots, c_m$. After deleting from $G$ all vertices of $A_j$ ($j \neq i$) except those on $q_1$ or $q_2$, the subgraph of $G$ corresponding to the block-path $p$ consists of the
path $q_1$, the block $A_i$, and the path $q_2$. After contracting $q_1$ and $q_2$, this subgraph consists of a block identical to $A_i$ except that it contains $c_0$ instead of $c_{i-1}$ and $c_m$ instead of $c_i$.

**Lemma 5.2.2** Let $G$ be series-parallel, and let $p = A_1, \ldots, A_m$ a compressible block-path of $G$. Then, $\text{cbp}(G,p) \approx_{\text{sp}} G$.

**Proof:** Let $A_i$ be block chosen by Compress Block-Path, and let $G_p$ be the subgraph of $G$ represented by $p$. $G_p$ is attached to the rest of $G$ by the cutpoints $c_0$ and $c_m$. $G' = \text{cbp}(G,p)$ has a block $A$ isomorphic to $A_i$ that is attached to the rest of $G'$ by $c_0$ and $c_m$. The graphs $G - V(G_p)$ and $G' - V(A)$ are identical. Thus, we can view Compress Block-Path as the substitution of $A$ for $G_p$ in $G$. Consider any $(H,L) \in \text{Env}(G)$. $H \circ_L G$ and $H \circ_L G'$ can be rewritten, respectively, as $J \circ_N G_p$ and $J \circ_N A$, where $N = [(u,c_0),(v,c_m)]$ for some distinct $u,v \in V(J)$. Thus, we need only show that $G_p$ and $A$ belong to the same $\approx_{\text{sp}}$-equivalence class with respect to coupling vertices $c_0$ and $c_m$.

Since $G$ is series-parallel, $G_p \notin \text{SP}1$. $G_p \in \text{SP}2$ if and only if $A_j \in \text{SP}2$ for some $1 \leq j \leq m$. By the choice of the block $A_i$, it follows that $G_p$ and $A$ both belong to either $\text{SP}2$ or $\text{SP}3$. □

**Definition 5.2.3** Let $A$ be any block of $G$. An edge $e = (u,v)$ of $A$ is **contractible** if (1) $u$ or $v$ is a non-coupling series vertex, or (2) $e$ is not redundant, $u$ and $v$ are non-coupling vertices each having degree at least 3, and $\{u,v\}$ does not separate $A$.

For a block $A$ of $G$, we denote by $\text{rb}(G,A)$ the result of applying the following operation:

**Reduce Block:** Repeatedly delete redundant edges of $A$ and contract contractible edges of $A$ until no such edges exist.
Let \( G' = \text{rb}(G, A) \), and let \( A' \) be the subgraph of \( G' \) corresponding to \( A \). By Lemma 5.1.1 and Theorem 5.1.1, \( A' \) is biconnected and series-parallel. Since \( A \) and \( A' \) have the same coupling vertices, it follows that \( A' \) is a block of \( G' \).

**Lemma 5.2.3** Let \( G \) be a series-parallel graph, and let \( A \) be any block of \( G \). Then, \( \text{rb}(G, A) \approx_{\text{sp}} G \).

**Proof:** Let \((H, L) \in \text{Env}(G)\). We show that \( \text{sp}(H \circ_L G) = \text{sp}(H \circ_L G') \). Let \( A' \) be the block of \( G' = \text{rb}(G, A) \) corresponding to \( A \). Since \( A' \) and \( A \) have the same coupling vertices, \( H \circ_L G \) and \( H \circ_L G' \) can be written, respectively, as \( J \circ A \) and \( J \circ A' \), where \( A \) and \( A' \) are glued onto \( J \) via their coupling vertices. Thus, we need only show that \( A' \approx_{\text{sp}} A \).

Consider \( e \in E(A) \). If \( e \) is redundant, then \( A - e \approx_{\text{sp}} A \) by Theorem 5.1.1. Suppose \( e \) is contractible. Then, either (1) \( e \) has an endpoint, say \( u \), that is a non-coupling series vertex, or (2) \( e \) is not redundant, and \( \{u, v\} \) does not separate \( A \). In the former case, \( u \) is also a non-coupling series vertex of \( J \circ_N A \) for any \((J, N) \in \text{Env}(A)\). Thus, since \( K_4 \) has no series vertices, \( A/e \approx_{\text{sp}} A \). In the latter case, \( A/e \approx_{\text{sp}} A \) by Theorem 5.1.2. Therefore, by the transitivity of replaceability, \( A' \approx_{\text{sp}} A \). \( \square \)

The burner is given in Figure 5.3. We shall now prove that the burner produces replaceable graphs.

**Theorem 5.2.1** For any graph \( G \), \( B_{\text{sp}}(G) \approx_{\text{sp}} G \).

**Proof:** Let \( G^b = B_{\text{sp}}(G) \). Suppose \( G \) is not series-parallel. Then, \( G^b \) consists of zero or more isolated pins and a graph homeomorphic from \( K_4 \). Thus, by Lemma 5.1.2, \( G^b \approx_{\text{sp}} G \). If \( G \) is series-parallel, then \( G^b \approx_{\text{sp}} G \) follows from the Lemmas 5.2.1, 5.2.2 and 5.2.3, and the transitivity of replaceability. \( \square \)
function Bsp(G: graph):graph
begin
  $G^b := G$;
  if $G^b$ has a subgraph $J$ homeomorphic from $K_4$ then
    Delete from $G^b$ each of its terminal vertices that is not in $J$;
    Contract an edge incident on each non-coupling series vertex of $G^b$;
  else
    while $G^b$ has a useless block $A$ do
      $G^b := db(G^b, A)$;
      for each maximal length compressible block-path $p$ in $G^b$ do
        $G^b := cbp(G^b, p)$;
      end;
      for each block $A$ of $G^b$ do
        $G^b := rb(G^b, A)$;
      end;
  end;
  return $G^b$
end

Figure 5.3: The series-parallel burner
It remains to show that the size of a burnt graph is linear in its number of pins. Let $G^b = B_{sp}(G)$. First, we compute the size of each biconnected component of $G^b$ by analyzing its triconnectivity tree. Then, an analysis of the bc-forest $bcf(G^b)$ will provide us with the desired result. While these techniques lead to straightforward proofs, they tend to overestimate the size of $G^b$. However, they are sufficient for our purposes.

**Lemma 5.2.4** Let $G$ be a biconnected series-parallel graph with $r > 0$ coupling vertices, and let $G'$ be the result of applying Reduce Block to $G$. Then, $G'$ has $O(r)$ vertices.

**Proof:** Let $T = tct(G')$. Each $v \in V(T)$ corresponds to a triconnected component $\beta(v)$ of $G'$. By assumption, $G'$ is series-parallel. Thus, every triconnected component of $G'$ is either a bond or a polygon. Let $P$ be the vertices of $T$ that correspond to polygons. Given the definition of Reduce Block, $G'$ contains no redundant edges and no contractible edges. Therefore, it follows that:

1. Every leaf of $T$ corresponds to a polygon.
2. Every non-coupling vertex of a polygon is the endpoint of a virtual edge.
3. Every real edge of a polygon has a coupling vertex endpoint.
4. Every edge of $G'$ whose endpoints are not coupling vertices belongs to a bond.

The two vertices of each bond are repeated in each of its neighboring polygons. Thus, $|V(G')|$ is bounded above by the sum of $|V(\beta(v))|$ over all $v \in P$.

Let $v \in P$, and assume $\beta(v)$ has $I_v$ virtual edges. The companion of a virtual edge in a polygon is a virtual edge in a bond, and vice-versa. Thus, since $T$ has one edge for each companion pair, the sum of $I_v$ over all $v \in P$ equals $|E(T)|$. Let
$S_v$ be the coupling vertices of $G'$ in $\beta(v)$ whose two incident edges are real. These vertices belong to no other polygons and no other bonds. Thus, the sum of $|S_v|$ over all $v \in P$ is at most $r$. Notice that every vertex in $V(\beta(v)) - S_v$ must be the endpoint of a virtual edge of $\beta(v)$. Thus, $|V(\beta(v))| \leq |S_v| + 2 \cdot I_v$. Therefore, $\sum_{v \in P} |V(\beta(v))| \leq r + 2 \cdot |E(T)|$.

It remains to show that $T$ has $O(r)$ edges. Let $R$ be the set of coupling vertices of $G'$. Let $v$ be any leaf of $T$. Then, since $\beta(v)$ is a polygon with only one virtual edge, it must have two real edges that share an endpoint, say $x$. Since no edge of $G'$ is incident on a non-coupling series vertex, $x$ is a coupling vertex. Furthermore, $x$ belongs to no other polygon or bond. Let $R_i$ consist of all such $x \in R$, and let $R_s = R - R_i$. Then, each leaf of $T$ can be matched with a unique member of $R_i$.

Consider a path $q = v_1, v_2, \ldots, v_m$ ($m > 0$) of series vertices in $T$. One of $v_1$ and $v_2$ belongs to $P$. Let it be $v_i$. Since $\beta(v_i)$ has only two virtual edges, it contains a real edge, and therefore, a vertex in $R_s$. If $i + 4 > m$, then $q$ has at most 5 vertices, one of which can be matched with a unique vertex in $R_s$. Assume $i + 4 \leq m$. Consider polygons $\beta(v_i)$, $\beta(v_{i+2})$, and $\beta(v_{i+4})$. Each contains a real edge with an endpoint in $R_s$. Since their real edges are distinct, at least two of their endpoints in $R_s$ must be distinct. Thus, a unique vertex in $R_s$ can be matched with at least every fifth vertex on $q$. Therefore, $|V(T)| \leq 2 \cdot |R_i| + 5 \cdot |R_s| \leq 5r$, and hence, $|E(T)| < 5r$. □

**Theorem 5.2.2** For any graph $G$ with $p$ pins, $B_{SP}(G)$ has $O(p)$ vertices and edges.

**Proof:** If $G$ is not series-parallel, then $B_{SP}(G)$ has at most $p + 4$ vertices and $p + 6$ edges. Assume $G$ is series-parallel, and let $G^b = B_{SP}(G)$. By Theorem 5.2.1, $G^b$ is also series-parallel. Since every series-parallel graph is planar, the number of edges of $G^b$ is linear in its number of vertices.
Consider the bc-forest \( F = bcf(G^b) \). An upper bound on \(|V(G^b)|\) is the sum of \( |V(\beta(x))| \) over all b-nodes \( x \) in \( F \). By Lemma 5.2.4, \(|V(\beta(x))| = O(s_x + t_x)\), where \( s_x \) is the number of pins in \( \beta(x) \) that are not cutpoints of \( G^b \), and \( t_x \) is the number of vertices of \( \beta(x) \) that are cutpoints of \( G^b \). The number of times that a cutpoint of \( G^b \) appears as a coupling vertex is exactly the degree in \( F \) of its corresponding c-node. The sum of the degrees of the c-nodes in \( F \) is equal to \(|E(F)|\). Thus, \(|V(G^b)| = O(p) + |E(F)|\). We show \(|E(F)| = O(p)\).

Consider a leaf \( b \) of \( F \). As a result of Delete Block, the block of \( G^b \) corresponding to \( b \) contains a pin that is not a cutpoint of \( G^b \). Consider any path \( p \) of series vertices in \( F \). As a result of Compress Block-Path, at least every fourth node on \( p \) corresponds to a block that contains a pin or a cutpoint that is a pin. Therefore, \(|V(F)| = O(p)|\), and hence, \(|E(F)| = O(p)|\). Thus, \(|V(G^b)| = O(p)|\). □

### 5.2.2 Run-time analysis

We now describe an implementation of the burner that operates in linear time. Let \( G \) be a graph. Suppose \( G \) is not series-parallel. Then, a \( O(\max(|V(G)|, |E(G)|)) \) algorithm by Liu and Geldmacher [25] can be used to extract a subgraph \( J \) of \( G \) that is homeomorphic from \( K_4 \). \( G^b \) consists of the pins in \( G - V(J) \) and a graph \( J' \) obtained from \( J \) by contracting an edge incident on each non-coupling series vertex. \( J' \) can be constructed in time linear in the size of \( J \). Thus, \( G^b \) can be constructed in time linear in the size of \( G \).

Suppose \( G \) is series-parallel. Depth-first search [34] can be used to locate and delete useless blocks of \( G \). The total time required for all Delete Block operations is linear in the size of \( G \). Depth-first search can also be used to locate maximal length compressible block-paths. The time required to compress a block-path \( p \) is linear...
in the size of the subgraph of $G$ represented by $p$. Since a block is involved in at most one application of Compress Block-Path, the time required to compress all compressible block-paths is linear in the size of $G$. It remains to show that Reduce Block has an implementation that operates in linear time.

Applied to a block $A$ of $G$, Reduce Block deletes redundant edges and contracts contractible edges until no such edges exist. Let $T$ be the triconnectivity tree of $A$. Since $A$ is series-parallel, it has at most $2 \cdot |V(A)| - 3$ edges [3]. Thus, by Theorem 2.1.1, $T$ has size $O(|V(A)|)$ and can be found in $O(|V(A)|)$ time. We show that a single post-order traversal of $T$ is sufficient to implement Reduce Block.

Every redundant edge of $A$ belongs to a bond corresponding to leaf of $T$, and every contractible edge belongs to a polygon. Let $v$ be the vertex of $T$ currently being visited. We consider whether or not $\beta(v)$ is a bond or a polygon.

Suppose $\beta(v)$ is a bond. If $v$ is an interior vertex of $T$, then $\beta(v)$ is not changed. If $v$ is a leaf, we delete all but one of the real edges of $\beta(v)$, and then merge it with its neighboring polygon.

Suppose $\beta(v)$ is a polygon. We traverse its edges in cyclic order, contracting a real edge $e$ if either (1) the next edge $f$ in $\beta(v)$ is real and $e$ and $f$ share an endpoint that is not a coupling vertex, or (2) both endpoints of $e$ are not coupling vertices. If $v$ is a leaf or a series node of $T$ (i.e. $\beta(v)$ has at most two virtual edges), then these contractions may reduce $\beta(v)$ to a bond. Assume that is the case. If $v$ is a leaf, it is handled as any other leaf whose triconnected component is a bond. If $v$ is a series node, we merge $\beta(v)$ and its two neighboring bonds, and delete any redundant real edges of the resulting bond.

Therefore, Reduce Block has an implementation that operates on $O(|V(A)|)$ time, and hence, the burner operates in linear time. By Theorems 3.1.1 and 3.1.2,
we have the following:

**Theorem 5.2.3** Applied to any hierarchical graph $\Gamma$, algorithm $BU$ and the burner $B_{SP}$ determine $sp(X(\Gamma))$ in $O(|\Gamma|)$ time.

## 5.3 Recognition of Outer-Planar Graphs

In this section, we present a hierarchical algorithm that, given $\Gamma$, determines $op(X(\Gamma))$ in linear time in the size of $\Gamma$. As in Section 5.2, we need only present the burner and prove that it produces replaceable graphs whose sizes are linear in their number of pins.

### 5.3.1 The burner

Every outer-planar graph is also series-parallel. As a result, the burner $B_{OP}$ is not much different from the series-parallel burner $B_{SP}$.

Suppose $B_{OP}$ is applied to a graph $G$ that is not outer-planar. The burner first determines if $G$ contains a subgraph homeomorphic from $K_4$. If such a subgraph $J$ exists, $B_{OP}$ returns a graph $G^b$ consisting of the pins in $G - V(J)$ and the graph $J'$ obtained from $J$ by contracting an edge incident on each non-coupling series vertex. If the burner finds that $K_4 \not\subseteq G$, it determines whether $G$ contains a subgraph homeomorphic from $K_{2,3}$. If such a subgraph $J$ exists, $B_{OP}$ returns a graph $G^b$ consisting of the pins in $G - V(J)$ and a graph $J'$ homeomorphic from $K_{2,3}$ obtained from $J$ by contracting as many edges as possible that are incident on non-coupling series vertices.

If $G$ is outer-planar, $B_{OP}$ first deletes all useless blocks from $G$ in exactly the same way as $B_{SP}$. It then proceeds to compress maximal length block-paths and
reduce the blocks of the graph using the Compress Block-Path and Reduce Block operations. However, these two operations are implemented differently from Bsp.

Lemma 5.1.4 motivates the changes to Compress Block-Path. Let \( A_1, \ldots, A_m \) be a compressible block-path of \( G \). The Compress Block-Path operation is as follows:

1. Find an \( i \) such that \( A_i \in \text{OP} 2 \). If none exists, find an \( i \) such that \( A_i \in \text{OP} 3 \). If none exists, find an \( i \) such that \( A_i \in \text{OP} 4 \). If none exists, go to step 3.

2. Find a simple path \( q_1 \) connecting \( c_0 \) and \( c_{i-1} \), and a simple path \( q_2 \) connecting \( c_i \) and \( c_m \). For each \( j \neq i \), delete from \( G \) all vertices in \( A_j \) except those on \( q_1 \) or \( q_2 \). Contract the edges of \( q_1 \) into \( c_0 \) and those of \( q_2 \) into \( c_m \). Return \( G \).

3. Find a simple path \( q \) connecting \( c_2 \) and \( c_m \). For each \( 2 < j \leq m \), delete from \( G \) all vertices in \( A_j \) except those on \( q \). Contract the edges of \( q \) into \( c_m \). Return \( G \).

Reduce Block is changed by restricting the edges that can be contracted. Let \( A \) be a block of \( G \) and \( e = (u, v) \in E(A) \):

- \( e \) is contractible if \( u \) and \( v \) are not coupling vertices, and either (1) \( u \) or \( v \) is a series vertex, or (2) \( e \) is not redundant, \( u \) and \( v \) each have degree at least 3, and \( \{u, v\} \) does not separate \( A \).

The edges contractible by Bsp but not by Bop are those edges whose endpoints consist of a non-coupling series vertex and a coupling vertex. We shall now prove that the burner Bop preserves replaceability.

Lemma 5.3.1 Let \( G \) be outer-planar, and let \( p = A_1, \ldots, A_m \) a compressible block-path of \( G \). Then, \( G' = \text{cbp}(G, A) \approx \text{op} \ G \).

Proof: Let \( G_p \) denote the subgraph of \( G \) corresponding to \( p \). We consider whether
Compress Block-Path executes step 2 or 3.

Suppose step 2 is executed. Let $A_i$ be the block chosen. Then, $G'$ has a block $A$ isomorphic to $A_i$. As in Lemma 5.2.2, we need only show that $G_p$ and $A$ are replaceable (where $c_0$ and $c_m$ are considered to be the coupling vertices). This follows from Lemma 5.1.4, and the choice of the block $A_i$.

Suppose step 3 is executed. Then, each $A_j$ belongs to $\approx_{op}$-equivalence class $OP5$. That is, $A_j$ consists of one or more edges with endpoints $c_{j-1}$ and $c_j$. In this case, $G_p$ is not replaceable with any single $A_j$, but is replaceable with any two blocks of $p$. Thus, since two blocks of $p$ remain in $G'$, it follows that $G' \approx_{op} G$. \[\]

Given Theorem 5.1.3, the following result can be proved in a manner analogous to the proof of Lemma 5.2.3.

**Lemma 5.3.2** For any outer-planar graph $G$ and any block $A$ of $G$, $rb(G, A) \approx_{op} G$.

The same techniques as used in the proof Theorem 5.2.1 provide the following result.

**Theorem 5.3.1** For any graph $G$, $B_{op}(G) \approx_{op} G$.

### 5.3.2 Run-time analysis

By Theorem 5.3.1, the burner produces replaceable graphs. It remains to show that every burnt graph has size linear in its number of pins. First, we determine the size of each biconnected component in terms of its number of coupling vertices, and then we analyze the size of the bc-forest.
Lemma 5.3.3 Let \( G \) be a biconnected outer-planar graph having \( r > 0 \) coupling vertices, and let \( G' \) be the result of applying Reduce Block to \( G \). Then, \( G' \) has \( O(r) \) vertices.

Proof: If \( r = 1 \), \( G' \) has at most two vertices. Assume \( r > 1 \). Then, \( G' \) is biconnected, outer-planar, and contains no redundant or contractible edges. Let \( H \) be the result of burning \( G \) according to the series-parallel version of Reduce Block. Given the definitions of contractible edges for the outer-planar and series-parallel Reduce Block operations, it follows that each vertex of \( G' \) that is not in \( H \) is a non-coupling series vertex adjacent to two distinct coupling vertices. Let \( S = \overline{V(G')} - \overline{V(H)} \), and let \( R \) be the set of coupling vertices of \( G' \). We show that \( |S| \leq r = |R| \). On the contrary, assume \( |S| > r \).

Let \( J \) be the bipartite subgraph of \( G' \) that has bipartition \( R \) and \( S \) and contains as many edges as possible. \( J \) has at least \( 2r + 1 \) vertices. Since each vertex of \( S \) is adjacent to two vertices of \( R \), \( J \) has at least \( 2(r + 1) \) edges. Therefore, \( J \) contains a cycle \( C \). Since \( J \) is bipartite, \( C \) has an even number of vertices, and since \( G' \) has no redundant edges, \( |V(C)| \geq 4 \).

Suppose \( C \) contains all vertices of \( R \). Since we assume \( |S| > r \), there is some \( v \in S - V(C) \) adjacent to distinct coupling vertices \( x \) and \( y \) of \( C \). But, since \( x \) and \( y \) are not adjacent on \( C \), \( K_{2,3} \subset J \subset G' \), a contradiction.

Suppose \( R \) has a vertex \( v \) not on \( C \). Since \( G' \) is biconnected, there are at least two paths that connect \( v \) to \( C \). These paths are vertex disjoint except for \( v \), and end on coupling vertices of \( C \) since the non-coupling vertices in \( C \) are series vertices of \( G' \). Thus, \( K_{2,3} \subset G' \), a contradiction.

Therefore, \( G' \) has at most \( r \) more vertices than \( H \), which, by Lemma 5.2.4, has \( O(r) \) vertices. \( \square \)
Theorem 5.3.2 For any graph $G$ with $p$ pins, $B_{op}(G)$ has $O(p)$ vertices and edges.

Proof: Virtually identical to that of Theorem 5.2.2. One need only note that at least every sixth node on a path of series nodes in $bcf(B_{op}(G))$ corresponds to a block that contains a pin or a cutpoint that is a pin. □

The implementation of $B_{op}$ is essentially the same as the implementation of $B_{sp}$. Syslo and Iri [33] provide a linear-time algorithm for determining if a graph is outer-planar, and Asano [3] provides a linear-time algorithm for extracting a subgraph homeomorphic from $K_{2,3}$, provided one exists. Thus, we have the following theorem:

Theorem 5.3.3 Applied to any hierarchical graph $Γ$, algorithm $B_U$ and the burner $B_{op}$ determine $op(X(Γ))$ in $O(|Γ|)$ time.

5.4 Extracting a Forbidden Subgraph

The problem we are now concerned with is the following: Given a hierarchical graph $Γ$, determine if $X(Γ)$ is series-parallel, and if it is not, generate the edges in a subgraph of $X(Γ)$ that is homeomorphic from $K_4$. We provide an algorithm GEN which solves this problem using work-space linear in the size of $Γ$. Similar techniques can be used to extract a subgraph homeomorphic from $K_{2,3}$.

By Theorem 5.2.1, for each $i$, $X(Γ_i)$ is series-parallel if and only if $G^b_i$ is series-parallel. It is easy to verify that the arguments in Section 5.2 imply that if $G^b_i$ contains a subgraph $H$ homeomorphic from $K_4$, then $X(Γ_i)$ has a subgraph $Z$ homeomorphic from $H$. Since $Γ$ has no useless cells, $X(Γ_i) ⊆ X(Γ)$, and hence, $Z ⊆ X(Γ)$. Algorithm GEN lists the edges in such a graph $Z$. 
5.4.1 The basic idea

Let $G$ be any graph. We shall first consider the problem of listing the edges in a subgraph of $G$ homeomorphic from a given subgraph of $G^b = B_{SP}(G)$. $G^b$ is a minor of $G$. Since we assume edges have names independent of their endpoints, each edge of $G^b$ is also an edge of $G$, although endpoints may differ. Recall from Section 2.1.3 that we consider an edge contraction as contracting an edge into one of its endpoints. Thus, every vertex of $G^b$ is also a vertex of $G$. It follows that for each $x \in V(G^b)$, there is a subgraph $T_x$ of $G$ that consists of all vertices and edges contracted into $x$ by the burner. We say $x$ simulates $T_x$. Notice that $T_x$ is always a tree — if contractions reduce the number of edges in a cycle to two, then one of the two remaining parallel edges will be deleted.

Consider any edge $e$ of $G^b$ incident on $x$. In $G$, $e$ is incident on some vertex of $T_x$ that we call the entry-point of $e$ into $T_x$. Entry-points allow us to maintain the simulated trees. Initially, a vertex of $G$ simulates itself. As edges are contracted, the simulated trees grow. Suppose $e$ is an edge with endpoints $x$ and $y$, and the entry-points of $e$ into $T_x$ and $T_y$ are $u$ and $v$, respectively. If we contract $e$ into $x$, we then connect $T_y$ to $T_x$ by adding $e$ between $u$ and $v$. This process is illustrated.
in Figure 5.4, where arrows point from edges to their entry-points.

For any tree $T$ and any nonempty subset $S$ of its vertices, there is a unique smallest connected subgraph of $T$ containing $S$. Thus, for any subgraph $H$ of $G^b$, we can construct a subgraph of $G$ homeomorphic from $H$ by expanding each vertex $w$ of $H$ as follows:

1. For $e \in E(F)$ incident on $w$, tag the entry-point of $e$ in $T_w$. If $\deg_F(w) = 1$, tag $w$ in $T_w$.

2. Root $T_w$ at any vertex, and find the least common ancestor $a$ of the tagged vertices. Let $T$ be the subtree of $T_w$ containing the unique path from $a$ to each tagged vertex. $T$ is the smallest connected subgraph of $T_w$ containing all tagged vertices.

3. Substitute $T$ for the vertex $w$ in $F$ by attaching each edge of $F$ incident on $w$ to its corresponding entry-point in $T$.

Expanding a vertex $w$ of $H$ takes time linear in the size of $T_w$ and the number of edges incident on $w$ in $H$. In Figure 5.5, we illustrate the expansion of two vertices of some $H \subseteq G^b$. One vertex has a single incident edge, while the other has three incident edges. Clearly, the graph obtained by expanding each vertex of $H$ is a subgraph of $G$ homeomorphic from $H$.

Until now, we have considered a special case of the generation problem in which no hierarchy was involved. The introduction of the hierarchy complicates matters. Let $\Gamma = (G_1, \ldots, G_n)$ be a hierarchical graph such that $X(\Gamma)$ is not series-parallel. Our goal is to list the edges in a subgraph of $X(\Gamma)$ that is homeomorphic from $K_4$.

For each $i$, $G^b_i$ is a minor of $\bar{G}_i$, and $\bar{G}_i$ is a minor of $X(\Gamma_i)$. Thus, a vertex $x$ of $G^b_i$ simulates a subgraph $S_x$ of $X(\Gamma_i)$. However, since $S_x$ may have size exponential
Figure 5.5: Expanding vertices
in the size of $\Gamma_i$, we do not store it explicitly. Instead, $x$ simulates a subgraph $T_x$ of $\bar{G}_i$, not $S_x$. That is, $S_x$ is distributed over the hierarchy: $x$ simulates $T_x$, each vertex of $T_x$ simulates another graph, and so on.

5.4.2 Algorithm GEN

Consider the graph $\bar{G}_i$ for some $i$. $\bar{G}_i$ is formed by replacing each nonterminal $l \cdot G_j$ of $G_i$ with the graph $G_j^b$. Thus, each edge of $\bar{G}_i$ comes from $G_i$ or comes from some $G_j^b$ by way of replacing some nonterminal $l \cdot G_j$. However, a vertex $v$ of $\bar{G}_i$ may come from $G_i$ or from some $G_j^b$ by replacing a nonterminal, or may be the result of many vertex identifications. Such vertices contribute to $v$ in $\bar{G}_i$.

A vertex of $G_i$ in $\bar{G}_i$ simulates only itself in $X(\Gamma_i)$. Thus, we need only concern ourselves with vertices that contribute to $v$ by nonterminal replacement. Let $M(v)$ be the set of all vertex and nonterminal pairs $(w, l \cdot G_j)$ such that $w \in V(G_j^b)$ contributes to $v$ by the replacement of $l \cdot G_j$. Then, $M(v)$ is nonempty unless $v$ was a vertex of $G_i$ adjacent to no nonterminals.

Consider the edges of $\bar{G}_i$ incident on $v$. If $M(v)$ is empty, then $v$ and these edges came from $G_i$. However, if $M(v)$ contains an element $(w, l \cdot G_j)$, then some of the edges incident on $v$ may have come from $G_j^b$ by the replacement of $l \cdot G_j$. The vertex $w$ and those edges constitute a subgraph of $G_j^b$.

Algorithm GEN is given in Figure 5.6. It applies the bottom-up method until some $G_i^b$ is found to contain a subgraph $H$ homeomorphic from $K_4$. If each $G_i^b$ is series-parallel, GEN indicates that $X(\Gamma)$ is series-parallel. Suppose $G_i^b$ is not series-parallel. Then, $G_i^b$ consists of a graph $H$ homeomorphic from $K_4$ and possibly some isolated pins. After extracting $H$ from $G_i^b$, GEN finds a pathname $\sigma$ from $G_n$ to $G_i$. Since $\Gamma$ has no useless cells, $\sigma$ can be found as follows:
Find the first $j > i$ such that $G_j$ has a nonterminal $l \cdot G_i$. Recursively find a pathname $\rho$ from $G_n$ to $G_j$. Return $\rho \cdot l$.

This pathname allows us to consider a single copy of $X(\Gamma_1)$ in $X(\Gamma)$. Having computed $\sigma$, GEN invokes a recursive procedure FIND-EDGES to list the edges in a subgraph $Z$ of $X(\Gamma)$ that is homeomorphic from $H$. FIND-EDGES is given in Figure 5.7. The following describes its operation.

The first step of the construction process is to expand each vertex of $H$ as defined in Section 5.4.1. This produces a subgraph $J$ of $\tilde{G}_i$ that is homeomorphic from $H$. The edges of $J$ are then listed. Each $v \in V(J)$ simulates a subgraph of $Z$ whose edges have not yet been listed. Let $E_v$ be the set of edges in $J$ incident on $v$. The subgraph of $Z$ simulated by $v$ is a combination of the subgraphs simulated by the vertices that contribute to $v$. For each $(w, l \cdot G_j)$ in $M(v)$, we list the edges in the subgraph of $Z$ simulated by $w$ as follows:

The vertex $w$ came from $G^h_j$ by replacing nonterminal $l \cdot G_j$. Thus, some of the edges in $E_v$ may have also come from $G^h_j$ by replacing $l \cdot G_j$. These edges and the vertex $w$ constitute a subgraph $T$ of $G^h_j$ that is a tree. If $T$ has no edges, then the subgraph of $Z$ simulated by $w$ is $w$ itself, and $T$ is discarded. Otherwise, FIND-EDGES is invoked recursively with parameters $T$, $\sigma \cdot l$, and $j$. However, only $w \in V(T)$ needs to be expanded, and the edges in $T$ do not need to be listed. We indicate this by marking each edge in $T$ and each vertex in $T$ except for $w$.

In the following section, we shall see that algorithm GEN need not explicitly maintain pathnames on vertices and edges. Instead, a system of pointers can be used to build pathnames when needed. The total space required for these pointers is linear in the size of $\Gamma$. 

Algorithm GEN

input: $\Gamma = (G_1, \ldots, G_n)$
output: "Series-Parallel" if $X(\Gamma)$ is series-parallel.

Otherwise, $E(J)$ for some $J \subseteq X(\Gamma)$ such that $K_4 \subseteq J$

for $i := 1$ to $n$ do

Construct $\tilde{G}_i$ as in algorithm BU;

$G_i^b := Bsp(\tilde{G}_i)$;

if $G_i^b$ is not series-parallel then

Extract $H \subseteq G_i^b$ homeomorphic from $K_4$;

Make each vertex and edge of $H$ unmarked;

Find a pathname $\sigma$ in $\Gamma$ from $G_n$ to $G_i$;

FIND-EDGES($H$, $\sigma$, $i$);

return

end

end;

return "Series-Parallel"

Figure 5.6: Algorithm GEN
procedure FIND-EDGES(H : graph, σ : pathname, i : integer)
begin
    Let J be the result of expanding each unmarked vertex of H;
    for each unmarked edge e of J do
        Find the pathname ρ of e in X(T_i);
        Output σ*ρ*e
    end;
    for each unmarked vertex v of J do
        for each (w, l*G_j) ∈ M(v) do
            Let T be the subgraph of G^l_j containing w and the edges
            of J incident on v that resulted from replacing l*G_j;
            Mark each edge of T and each vertex of T except for w;
            if |E(T)| > 0 then
                FIND-EDGES(T, σ*l, j);
            end
        end
    end
end

Figure 5.7: Procedure FIND-EDGES
5.4.3 Implementation details

In this section, we describe the modifications to algorithm BU and the series-parallel burner needed for algorithm GEN.

Changes to the burner are minimal: whenever Reduce Block or Compress Path contracts an edge, the trees simulated by the endpoints of the edge must be connected together as described in Section 5.4.1.

The extensions to algorithm BU are also relatively minor. Let \( \Gamma = (G_1, \ldots, G_n) \). For each \( i \), we assume algorithm GEN keeps a copy of the graphs \( G_i \), \( \tilde{G}_i \), and \( G_i^b \). In Section 5.4.2, we made some simplifying assumptions regarding the operation of procedure FIND-EDGES. These are addressed below:

1. For each \( v \in V(\tilde{G}_i) \), each element \( (w, l \cdot G_j) \) of \( M(v) \) can be represented by two pointers: one to the vertex \( w \) of \( G_j^b \), and the other to the nonterminal \( l \cdot G_j \) of \( G_i \).

2. For each \( e \in E(\tilde{G}_i) \), FIND-EDGES can determine the pathname of \( e \) in \( X(\Gamma_i) \) using the following data:
   
   - A pointer from each edge of \( G_i^b \) to the corresponding edge of \( \tilde{G}_i \).
   - Two pointers from each \( e \in E(\tilde{G}_i) \). Edge \( e \) comes from \( G_i \) or comes from some \( G_j^b \) by replacing a nonterminal \( l \cdot G_j \). One pointer points to the corresponding edge of \( G_i \) or \( G_j^b \). If \( e \) comes from \( G_i \), the second pointer is null. Otherwise, it points to the nonterminal \( l \cdot G_j \) of \( G_i \).

The pathname for \( e \) in \( X(\Gamma_i) \) can be found as follows: Examine the two pointers from \( e \) to determine the graph it came from. If \( e \) is from \( G_i \), return \( e \), the empty string. If \( e \) is from \( G_j^b \) via nonterminal \( l \cdot G_j \), follow the pointer from \( e \) in \( G_j^b \).
to \( e \) in \( \tilde{G}_j \), and recursively find its pathname \( \rho \) in \( X(\Gamma_j) \). Return \( \rho \ast l \).

These systems of pointers can be initialized when \( \tilde{G}_i \) is constructed and maintained as the graph is burned. In each case, the total number of pointers in linear in the size of \( \tilde{G}_i \), which, by Theorem 5.2.2, is linear in the size of \( G_i \). The trees simulated by vertices of \( G^b_i \) are disjoint subgraphs of \( \tilde{G}_i \), and hence, require total space linear in \( |G_i| \). Thus, the work-space required for each \( i \) is linear in the size of \( G_i \), and the total work-space required is linear in the size of \( \Gamma \).

5.5 Summary

In this chapter, we presented linear-time hierarchical algorithms that determine whether or not the expansion of a hierarchical graph \( \Gamma \) is series-parallel or outer-planar. The basic techniques used by the burners were edge deletion and contraction. As a result, the burnt graph \( G^b \) is small, replaceable minor of \( G \) such that, for any \( (H,L) \in \text{Env}(G) \), if \( F \subseteq H \circ_L G^b \) is homeomorphic from \( K_4 \), then \( H \circ_L G \) contains a subgraph homeomorphic from \( F \). Based on this observation, we devised an algorithm that generates the edges of a subgraph of \( X(\Gamma) \) homeomorphic from \( K_4 \), provided one exists. The algorithm uses work-space linear in the size of \( \Gamma \).
6. MATROID OPTIMIZATION PROBLEMS

As there seems to be no correlation between the hierarchical and non-hierarchical complexities of problems, a more fruitful approach might be to look for families of problems for which efficient hierarchical algorithms exist. In this chapter, we characterize two such families of problems. Both are problems of finding the cost of an optimum base of a matroid defined on the edges of a graph. Their common characteristic is that the circuits of these matroids are the edge sets of graphs homeomorphic from members of a finite set of graphs. These matroids were studied by Simões Pereira [32], Matthews [26,27], and White and Whiteley [39], and include the graphic matroid, whose bases are the edges of spanning forests of a graph. Our results generalize some of Lengauer's work on minimum spanning forests [19] and also, we believe, puts it in a new perspective by exhibiting to what extent his ideas work for other problems.

The chapter is organized as follows. Matroid definitions and concepts are presented in Section 6.1. In Section 6.2, we develop a nondeterministic hierarchical algorithm that computes optimum bases of certain classes of matroids defined on the edges of graphs. This algorithm serves as a foundation for the other algorithms presented in this chapter. In Section 6.3, we describe the two families of matroid classes discussed above, and give polynomial-time hierarchical algorithms for computing costs of optimum bases of these matroids when defined $E(X(\Gamma))$. In Section 6.5, we
develop space-efficient hierarchical algorithms for generating optimum bases instead of their costs. A brief summary and a discussion of some further results is given in Section 6.6.

6.1 Matroids and The Greedy Algorithm

6.1.1 Definitions

A matroid $M = (S, I)$ is a finite set $S$ called the domain, and a family $I$ of subsets of $S$, called independent sets, that satisfy the following axioms [38]:

(M1) $\emptyset \in I$.

(M2) If $A \in I$ and $B \subseteq A$, then $B \in I$.

(M3) For any $A \subseteq S$, all maximally independent subsets of $A$ have the same cardinality.

Maximally independent subsets of $S$ are called bases of $M$. Subsets of $S$ not in $I$ are called dependent, and minimally dependent subsets are called circuits of $M$. Every dependent subset of $S$ contains at least one circuit. The circuits of $M$ satisfy the following axioms [38]:

(C1) No circuit is a proper subset of another circuit.

(C2) If $X$ and $Y$ are circuits, $X \neq Y$, and $e \in X \cap Y$, then $M$ has a circuit $Z \subseteq (X \cup Y) - e$.

A set is independent in a matroid if and only if it contains no circuit. Any matroid is uniquely determined by its bases and also by its circuits [38].
Let $W$ be the set of bases of $M = (S, I)$. Then, the set $W^* = \{S - A : A \in W\}$ is the set of bases of a matroid $M^*$ on $S$ called the dual of $M$. The circuits of $M^*$ are called cocircuits of $M$.

**Fact 6.1.1** [38] The cocircuits of $M = (S, I)$ are those $C \subseteq S$ such that $C$, but no proper subset of $C$, has at least one element in common with every base of $M$.

Let us examine some well-known matroids. Let $S$ be a finite set with $q$ elements, and let $0 \leq m \leq q$. The independent sets of the uniform matroid [38] $U_{m,q}$ on $S$ are all subsets of $S$ having at most $m$ elements. Let $G$ be a directed graph. The independent sets of head partition matroid [30] on $G$ consist of all subsets of $E(G)$ in which no two arcs have the same head. The cycle or graphic matroid [38] on an undirected graph $G$ is a matroid $M = (E(G), I)$ where for any $A \subseteq E(G)$, $A \in I$ if and only if the subgraph of $G$ induced by $A$ is a forest. The bases of $M$ are the spanning forests of $G$, and the circuits of $M$ are the cycles of $G$. We will revisit cycle matroids in Section 6.3.

**Definition 6.1.1** Let $M = (S, I)$ be a matroid, and let $A \subseteq S$.

- $M \uparrow A$, the restriction of $M$ to $A$, is a matroid on $A$ whose independent sets are the subsets of $A$ that are independent in $M$.

- $M - A$, the result of deleting $A$ from $M$, is the matroid $M \uparrow (S - A)$.

- $M/A$, the result of contracting $A$ from $M$, is a matroid with domain $S - A$. For $W \subseteq S - A$, $W$ is an independent set of $M/A$ if and only if $W \cup B \in I$ for some base $B$ of $M \uparrow A$. 
The independence of $W$ in $M/A$ does not depend on any particular base $B$ of $M \uparrow A$ [5]. That is, the phrase "for some base" can be replaced by "for every base".

6.1.2 Properties of matroids

**Theorem 6.1.1** [5] Let $M = (S, I)$ be a matroid and let $X$ and $Y$ be disjoint subsets of $S$. Then, $(M/X)/Y = M/(X \cup Y)$, $(M - X) - Y = M - (X \cup Y)$, and $(M - X)/Y = (M/Y) - X$.

A minor of a matroid $M$ is the matroid obtained by a sequence of deletions and contractions from $M$. By the preceding theorem, any minor of $M$ can be written in the form $(M - D)/C$ where $D$ is the set of all deleted elements and $C$ is the set of all contracted elements. We now present some basic facts on matroids. Let $M = (S, I)$ be a matroid.

**Fact 6.1.2** [38] Let $B$ be a base of $M$. Then, for any $e \in S - B$, $B \cup e$ contains a unique circuit $X$, and $e \in X$. For any $f \in X$, $(B \cup e) - f$ is a base of $M$.

**Fact 6.1.3** [38] For any $e \in S$, the circuits of $M - e$ are precisely the circuits of $M$ that do not contain $e$.

**Fact 6.1.4** [5] Let $C$ be the set of circuits of $M$. For any $e \in S$, the circuits of $M/e$ are the minimal members of $\{X - e : X \in C$ and $X - e \neq \emptyset\}$.

**Fact 6.1.5** For any $e \in S$, the cocircuits of $M/e$ are precisely the cocircuits of $M$ that do not contain $e$.

**Proof:** The cocircuits of a matroid are the circuits of its dual. But, $(M/e)^* = M^* - e$ [38], and by Fact 6.1.3, the circuits of $M^* - e$ are the circuits of $M^*$ not containing $e$. □
6.1.3 A variant of the greedy algorithm

Let $M = (S, I)$ be a matroid with an integer or real-valued cost function $c$ defined on its domain $S$. $B \subseteq S$ is said to be an optimum base of $M$ if $B$ is a base of $M$ and $c(B) = \sum_{b \in B} c(b)$ is a minimum. The minimum-cost base problem is to find an optimum base of $M$. This problem is solved by the Greedy Algorithm \cite{38}, which constructs an optimum base $B$ of $M = (S, I)$ as follows:

$$B := \emptyset. \text{ While } B \text{ is not a base of } M, \text{ add to } B \text{ a lowest cost element } e \in S - B \text{ such that } B \cup e \in I.$$  

The greedy algorithm will also find a maximum cost base of $M$ by simply changing the phrase "lowest cost" to "highest cost".

Lawler \cite{18} introduced a variant of the greedy algorithm that relies on deletion and contraction of matroid elements. The algorithm, which we call L-Greedy, is nondeterministic in that the deletion and contraction operations can be applied in any order. Let $M = (S, I)$ be a matroid and $e \in S$. Then, $e$ can be deleted (contracted) from $M$ by L-Greedy provided it satisfies property LG1 (LG2) given below:

(LG1) $e$ is a highest cost element in some circuit of $M$.

(LG2) $e$ is a lowest cost element in some cocircuit of $M$.

We shall write L-Greedy using a do-loop of guarded commands, as described by Dijkstra \cite{6}. Such a do-loop is of the form

$$\textsf{do } g_1 \to S_1 \[ g_2 \to S_2 \[ \cdots \[ g_m \to S_m \textsf{ od}$$

where each $g_i$ is a boolean expression called a guard, and each $S_i$ is a sequence of statements. At the start of each loop iteration, all guards are evaluated. If no guards
Algorithm L-Greedy

input: a matroid $M$
output: an optimum base $F$ of $M$

$F := \emptyset$;

do
  $M$ has an element $e$ satisfying LG1 $\rightarrow M := M - e$;
  $M$ has an element $e$ satisfying LG2 $\rightarrow M := M/e$; $F := F \cup e$;
  od;

return $F$

Figure 6.1: Algorithm L-Greedy

are true, the loop terminates. Otherwise, among the guards that are true, one is chosen and its attached statements are executed. L-Greedy is given in Figure 6.1.

Theorem 6.1.2 [18] L-Greedy returns an optimum base of any matroid.

L-Greedy creates minors when applied to a matroid $M$. By Theorem 6.1.1, the minors are of the form $(M - D)/C$ for some disjoint sets $C$ and $D$. We say that the matroid $(M - D)/C$ is obtainable by L-Greedy from $M$ if there is some valid execution sequence of L-Greedy in which $(M - D)/C$ is formed at some intermediate step.

By Theorem 6.1.2, L-Greedy always computes optimum bases. The following lemma proves that any optimum base can be found by L-Greedy.

Lemma 6.1.1 Let $M = (S, I)$ be a matroid with cost function $c$, and let $F$ be any optimum base of $M$. Then, there is a valid execution sequence of L-Greedy that
returns \( F \).

**Proof:** Let \( D = S - F \). We show that \((M - D)/F\), which has an empty domain, is obtainable by L-Greedy, in which case L-Greedy returns \( F \).

Let \( e \in D \), and consider \( F \cup e \). By Fact 6.1.2, \( e \) belongs to the unique circuit \( X_e \) of \( F \cup e \), and \((F \cup e) - f \) is a base of \( M \) for any \( f \in X_e \). Suppose for some \( f \in X_e \), \( c(f) > c(e) \). Then, \((F \cup e) - f \) is a base of \( M \) of lower cost than \( F \), a contradiction. Hence, \( e \) is a highest cost element of \( X_e \). Thus, for any \( e \in D \), there is a circuit \( X_e \subseteq F \cup e \) of \( M \) in which \( e \) is a maximum cost element. By Fact 6.1.3, for any distinct \( e, f \in D \), \( X_f \) is a circuit of \( M - e \) and \( X_e \) is a circuit of \( M - f \). Thus, \( M - D \) is obtainable by L-Greedy from \( M \).

\( F \) is the domain of the matroid \( M - D \), as well as its only base. Then, by Fact 6.1.1, each \( e \in F \) is by itself a cocircuit of \( M - D \). By Fact 6.1.5, for any distinct \( e, f \in F \), \( \{f\} \) is a cocircuit of \((M - D)/e\), and \( \{e\} \) is a cocircuit of \((M - D)/f\). Thus, \((M - D)/F \) is obtainable by L-Greedy from \( M - D \). \( \square \)

The following is a restatement of Theorem 6.1.2 and Lemma 6.1.1 in terms of the deletion and contraction operations used by L-Greedy.

**Corollary 6.1.1** For any matroid \( M = (S, I) \), any \( e \in S \), and any \( W \subseteq S - e \):

1. *If \( e \) satisfies LG1, then \( W \) is an optimum base of \( M - e \) if and only if \( W \) is an optimum base of \( M \).*

2. *If \( e \) satisfies LG2, then \( W \) is an optimum base of \( M/e \) if and only if \( W \cup e \) is an optimum base of \( M \).*

**Proof:** If \( e \) satisfies LG1, then \( M \) has a circuit in which \( e \) is of highest cost, and hence, \( M - e \) is obtainable by L-Greedy. Since no elements have been contracted
in forming $M - e$ from $M$, it follows by Theorem 6.1.2 and Lemma 6.1.1 that any optimum base of $M - e$ is also an optimum base of $M$. By Definition 6.1.1, any base of $M$ that does not contain $e$ is also a base of $M - e$.

If $e$ satisfies LG2, then $M$ has a cocircuit in which $e$ is of lowest cost, and hence, $M/e$ is obtainable by L-Greedy. Let $W \subseteq S - e$ be any optimum base of $M/e$. By Lemma 6.1.1, L-Greedy can compute $W$ when applied to $M/e$. Therefore, it follows by Theorem 6.1.2 that $W \cup e$ is an optimum base of $M$. Let $W \subseteq S - e$ be such that $W \cup e$ is an optimum base of $M$. Then, by Definition 6.1.1, $W$ is a base of $M/e$. Suppose $W$ is not an optimum base of $M/e$. Then for any optimum base $W'$ of $M/e$, $W' \cup e$ is an optimum base of $M$ of lower cost than $W \cup e$, a contradiction. □

6.2 A Greedy Algorithm for Hierarchical Graphs

Let $\mathcal{M}$ be a class of matroids defined on graphs such that, for any graph $G$, $\mathcal{M}(G)$ is a matroid on $E(G)$ that satisfies the following property:

(D-Closure) for any $e \in E(G)$, $\mathcal{M}(G - e) = \mathcal{M}(G) - e$.

We provide an algorithm H-Greedy that, given $\Gamma$, returns an optimum base of $\mathcal{M}(X(\Gamma))$. Matroid classes satisfying D-Closure are discussed in Section 6.3.

6.2.1 The burner

H-Greedy manipulates pairs of the form $(G, E)$, where $G$ is a graph and $E$ is a set of edges disjoint from $E(G)$. The graph and edge set of a pair $R$ will be denoted by $R.G$ and $R.E$, respectively. The environment of $R$, Env($R$), is the set of all $(S, L)$ such that $(S.G, L) \in$ Env($R.G$). Pair composition is defined as follows: for $(S, L) \in$ Env($R$), $R \circ_L S = (R.G \circ_S L.S.G, R.E \cup S.E)$. 
117

The burner is a function $B$ that takes a pair $(G, E)$ and returns a pair $B((G, E)) = (G^b, E \cup C)$, where $G^b$ is a minor of $G$ obtained from some subgraph $J$ of $G$ by contracting $C \subseteq E(J)$. The burner is closely related to algorithm L-Greedy. An edge $e$ of a graph $G$ may be deleted (contracted) provided it satisfies condition HG1 (HG2) given below:

(HG1) $e$ is a highest cost element in some circuit of $\mathcal{M}(G)$.

(HG2) $\text{Env}(G) = \text{Env}(G/e)$ and, for every $(H, L) \in \text{Env}(G)$,

- $\mathcal{M}(H \circ_L (G/e)) = \mathcal{M}(H \circ_L G)/e$, and
- $e$ is a lowest cost element in some cocircuit $X \subseteq E(G)$ of $\mathcal{M}(H \circ_L G)$.

The burner is nondeterministic, and is described in Figure 6.2 using guarded commands. For some matroid classes, it may be difficult to find all edges satisfying HG2. Thus, the last guarded command allows the burner to terminate after no circuits remain, but before all possible contractions have been made. Isolated terminals are deleted because they are not involved in the gluing process and have no bearing on the matroids we consider.

6.2.2 Algorithm H-Greedy

Algorithm H-Greedy is an instance of algorithm GBU defined as follows. Let $\mathcal{G}$ be the set of all graphs with edge costs. Let $\langle \mathcal{E}, \cup, \emptyset \rangle$ be the algebra in which the set $\mathcal{E}$ consists of all subsets of the edge sets of graphs in $\mathcal{G}$. Let the solution space $\mathcal{S}$ be $\mathcal{E}$. Then, $\mathcal{T} = \mathcal{G} \times \mathcal{E}$ is the set of all pairs manipulated by the burner, and $(\mathcal{T}, \circ)$ is a gluing system. For $(G, E) \in \mathcal{T}$, define the solution mapping $\Phi$ as follows:

$$\Phi((G, E)) = \{A \cup E : A \text{ is an optimum base of } \mathcal{M}(G)\}.$$
function \texttt{B}((G, E) : pair):pair
begin
  \begin{align*}
    & H := G; C := E; \\
    & \textbf{do} \quad e \in E(H) \text{ satisfies } H_1 \rightarrow H := H - e; \\
    & \quad \textbf{if} \quad e \in E(H) \text{ satisfies } H_2 \rightarrow H := H/e; C := C \cup e; \\
    & \quad \textbf{if} \quad \mathcal{M}(H) \text{ contains no circuits } \rightarrow \text{ exit do-loop} \\
    & \textbf{od}; \\
    & \text{Delete any isolated terminals from } H; \\
    & \text{return } (H, C)
  \end{align*}
end

Figure 6.2: The H-Greedy burner

For a graph \( G \), \( \mathcal{M}(G) \) may have many optimum bases. Thus similarity, which we denote by \( \sim_M \), is not necessarily an equivalence relation.

For a pair \( (G, E) \), let \( s*(G, E) = (s*G, s*E) \). We assume algorithm H-Greedy uses the prefix operation when gluing tuples, thus ensuring that pathnames are maintained on the graphs and edge sets within pairs. Let \( \mathcal{M} \) a matroid class satisfying D-Closure.

The following lemma is trivial, yet crucial to the results presented here.

**Lemma 6.2.1** Let \( G \) be a graph and \( (H, L) \in \text{Env}(G) \). Then, for any \( e \in E(G) \),
\[
H \circ_L (G - e) = (H \circ_L G) - e.
\]
If, in addition, the endpoints of \( e \) are not distinct pins, then
\[
\text{Env}(G/e) = \text{Env}(G) \quad \text{and} \quad H \circ_L (G/e) = (H \circ_L G)/e.
\]

**Lemma 6.2.2** For any pair \( R, B(R) \sim_M R \).

**Proof:** Let \( R = (G, E) \), and let \( e \in E(G) \) satisfy HG1 or HG2. We show that \( R' \sim_M R \), where \( R' \) is obtained from \( R \) by applying the appropriate guarded statements of
B. Then, the lemma follows by transitivity. Let \( S = \langle H, A, L \rangle \in \text{Env}(R) \). Note that the set \( A \) in the pair \( S \) has no bearing on the similarity of \( R' \) with \( R \).

**Case 1:** \( e \) satisfies HG1.

Then, \( R' = (G - e, E) \). By HG1, \( M(G) \) contains a circuit \( X \) in which \( e \) is a highest cost element. Since \( G \) is a subgraph of \( H \odot L \), it follows by D-Closure that \( M(H \odot L \ G) \uparrow E(G) = M(G) \). Thus, every circuit of \( M(G) \) is also a circuit of \( M(H \odot L \ G) \). Then, by Corollary 6.1.1, \( W \subseteq E(H \odot L \ G) - e \) is an optimum base of \( M(H \odot L \ G) - e \) if and only if \( W \) is an optimum base of \( M(H \odot L \ G) \). By Lemma 6.2.1 and property D-Closure, \( M(H \odot L (G - e)) = M(H \odot L \ G) - e \). Thus, \( \Phi(S \odot L R') \subseteq \Phi(S \odot L R) \).

**Case 2:** \( e \) satisfies HG2.

Then, \( R' = (G/e, E \cup e) \). By property HG2, \( M(H \odot L (G/e)) = M(H \odot L \ G)/e \), and \( M(H \odot L \ G) \) has a cocircuit \( X \subseteq E(G) \) in which \( e \) is of lowest cost. Then, by Corollary 6.1.1, for any \( W \subseteq E((H \odot L \ G)/e) \), \( W \) is an optimum base of \( M(H \odot L \ G)/e \) if and only if \( W \cup e \) is an optimum base of \( M(H \odot L \ G) \). By Lemma 6.2.1, \( (H \odot L \ G)/e = H \odot L (G/e) \). Hence, \( \Phi(S \odot L R') \subseteq \Phi(S \odot L R) \).

We now prove the correctness of H-Greedy.

**Theorem 6.2.1** If \( M \) satisfies D-Closure, then applied to any hierarchical graph \( \Gamma = (G_1, \ldots, G_n) \), algorithm H-Greedy returns an optimal base of \( M(X(\Gamma)) \).

**Proof:** By Lemma 6.2.2, the burner preserves similarity, and hence, by Theorem 3.2.1, H-Greedy returns an element of \( \Phi(T_n) \), where \( T_n = (X(\Gamma), \emptyset) \). By definition, \( \Phi(T_n) \) is the set of all optimum bases of \( M(X(\Gamma)) \). \( \Box \)
6.3 Matroidal Families

In this section, we describe matroids on graphs whose circuits can be described by subgraphs homeomorphic to a member of a finite set of graphs.

The following definition is given by Matthews in [27].

**Definition 6.3.1** A nonempty family of graphs \( C \) is called a **matroidal family** if, for any graph \( G \), the subgraphs of \( G \) isomorphic to members of \( C \) are the circuits of a matroid on \( E(G) \) (i.e., they satisfy axioms C1 and C2). A matroidal family \( C \) is **homeomorphic** if it is closed under homeomorphism: \( G \in C \) and \( H \) homeomorphic to \( G \) implies \( H \in C \).

Let \( C \) be a homeomorphic matroidal family. If \( C' \subseteq C \) contains a minimal member of each equivalence class of \( C \) under homeomorphism, then \( C \) is the set of all graphs homeomorphic from a member of \( C' \).

Simões Pereira [32] found that the only homeomorphic matroidal families of finite, connected graphs are (1) all cycles — graphs homeomorphic from a loop, and (2) all bicycles — graphs homeomorphic from one of the graphs given in Figure 6.3. The cycle matroid of a graph \( G \) is the matroid whose circuits are the edges of cycles in \( G \). The **bicircular matroid** [27] of \( G \) is the matroid whose circuits are the edges of bicycles in \( G \).

**Definition 6.3.2** [27] A graph contains \( k \) independent cycles (\( k \) independent bicycles) if the deletion of \( k \) edges is necessary and sufficient to produce a graph containing
no cycles (bicycles). A graph is \textit{k-cycle-minimal} (\textit{k-bicycle-minimal}) if it contains \( k \) independent cycles (bicycles), but no proper subgraph of it does. Let \( C_k \) (\( B_k \)) denote the set of all \( k \)-cycle-minimal (\( k \)-bicycle-minimal) graphs.

For any \( k > 0 \), it can be shown that both \( C_k \) and \( B_k \) have a finite number of equivalence classes under homeomorphism. Thus, each of these matroid classes can be described by a finite number of homeomorphism-minimal graphs.

Matthews [27] characterized the homeomorphic matroidal families of finite (possibly unconnected) graphs as follows:

\textbf{Theorem 6.3.1} The only homeomorphic matroidal families of finite graphs are the families \( C_k \) and \( B_k \) for each positive integer \( k \).

For a graph \( G \) and an integer \( k > 0 \), let \( C_k(G) \) (\( B_k(G) \)) denote the matroid on \( E(G) \) whose circuits are the subgraphs of \( G \) isomorphic to a member of \( C_k \) (\( B_k \)). Then, \( C_1(G) \) is the cycle matroid of \( G \), and \( B_1(G) \) is the bicircular matroid of \( G \).

In the next section, we shall use algorithm H-Greedy to construct efficient hierarchical algorithms for determining costs of optimum bases of \( C_k(X(\Gamma)) \) or \( B_k(X(\Gamma)) \). To use H-Greedy, we must show that these matroids classes satisfy the D-Closure.

\textbf{Lemma 6.3.1} For any graph \( G \), any \( e \in E(G) \), and any \( k > 0 \), \( B_k(G - e) = B_k(G) - e \) and \( C_k(G - e) = C_k(G) - e \).

\textbf{Proof:} We consider only \( B_k(G) \). The proof is is essentially identical for \( C_k(G) \). The \( k \)-bicycle-minimal subgraphs of \( G - e \) are the \( k \)-bicycle-minimal subgraphs of \( G \) that do not contain \( e \). By Fact 6.1.3, the circuits of \( B_k(G - e) \) are the circuits of \( B_k(G) \) that do not contain \( e \). Thus, \( B_k(G - e) \) and \( B_k(G) - e \) have the same circuits, and therefore, are the same matroid. \( \Box \)
Simply deleting edges will not make our algorithms efficient because the size of an optimum base of a matroid on $X(\Gamma)$ can be as large as $X(\Gamma)$ itself. Thus, our burner must also contract edges that satisfy HG2. The following lemma describes edges $e$ of a graph $G$ for which $B_k(G/e) = B_k(G)/e$ and $C_k(G/e) = C_k(G)/e$.

**Lemma 6.3.2** Let $G$ be a graph, $e \in E(G)$, and let $k > 0$. If $e$ is a pendant edge or is in series with an edge $f \in E(G)$, then $C_k(G/e) = C_k(G)/e$ and $B_k(G/e) = B_k(G)/e$. In addition, if $e$ is an isolated loop of $G$, then $B_k(G/e) = B_k(G)/e$.

**Proof:** We consider only $B_k(G)$ — the proof is similar for $C_k(G)$. The circuits of $B_k(G)$ are the edges of $k$-bicycle-minimal subgraphs of $G$. We show that in each case, $B_k(G/e)$ and $B_k(G)/e$ have the same circuits.

Suppose $e$ is a pendant edge or an isolated loop of $G$. No $k$-bicycle-minimal graph has an isolated loop or pendant edge. Thus, $e$ belongs to no circuits of $B_k(G)$, and hence, the circuits of $B_k(G)$ are the circuits of $B_k(G/e)$. By Fact 6.1.4, the circuits of $B_k(G/e)$ are the circuits of $B_k(G)$.

Suppose $e$ is in series with $f$. Then, they belong to exactly the same cycles in $G$, and hence, to the same circuits of $B_k(G)$. Therefore, since the class of graphs $B_k$ is closed under homeomorphism, for any subgraph $X$ of $G$ containing $e$, $X$ is $k$-bicycle-minimal if and only if $X/e$ is $k$-bicycle-minimal. Thus, the circuits of $B_k(G/e)$ are $\{E(X/e) : X$ is a $k$-bicycle-minimal subgraph of $G\}$. This is precisely the set $Q = \{Y - e : Y$ is a circuit of $B_k(G)\}$. Thus, every member of $Q$ is minimal, and therefore, by Fact 6.1.4, $Q$ is the set of circuits of $B_k(G)/e$. $\square$
6.4 Cost-Finding Algorithms

We now present algorithms for determining costs of optimum bases of the matroids $B_k(X(\Gamma))$ and $C_k(X(\Gamma))$, for any $k > 0$ and any hierarchical graph $\Gamma$. For a graph $G$, let $\mathcal{M}(G)$ denote the matroid $B_k(G)$ or $C_k(G)$ for some $k > 0$.

The algorithm, called algorithm Base-Cost, is a straightforward modification of algorithm H-Greedy. Instead of pairs consisting of a graph and a set of edges, Base-Cost manipulates pairs $R$ containing a graph $R.G$ and a number $R.w$. $R.w$ records the cost of a set of edges, instead of the edges themselves. As we are concerned only with costs, the prefix operation is not used by Base-Cost. The absence of pathnames on the edges and vertices of graphs saves time and space. The solution mapping for a pair $\langle G, w \rangle$ is as follows:

$$\Phi(\langle G, w \rangle) = \{c - w : c \text{ is the cost of an optimum base of } \mathcal{M}(G)\}.$$ 

The burners for each of the classes $B_k$ and $C_k$ are identical except for their first step, which requires an algorithm for computing an optimum base of the corresponding matroid on a non-hierarchical graph. The burner is given in Figure 6.4.

There is a close relationship between the Base-Cost burner and the H-Greedy burner. STEP 1 is equivalent to deleting edges satisfying condition HG1 until no circuits remain. As a result, $E(G^b)$ is an independent set of $\mathcal{M}(G^b)$, and hence, $\Phi(\langle G^b, w \rangle) = \{c(E(G^b)) + w\}$. We shall see that STEP 2 is equivalent to repeatedly contracting edges that satisfy condition HG2.

6.4.1 Correctness of algorithm Base-Cost

Let $\Gamma$ be a hierarchical graph. In light of Theorem 6.2.1 and Lemma 6.3.1, we need only show that the edges contracted by the burner satisfy condition HG2 to
function B((G, w) : pair):pair
begin
  STEP 1:
  Find $H \subseteq G$ such that $V(H) = V(G)$ and $E(H)$ is an optimum base of $\mathcal{M}(G)$;
  STEP 2:
  for each pendant edge $e$ with a terminal pendant vertex do
    $w := w + c(e)$;
    $H := H/e$
  end;
  for each pair of edges $e$ and $f$ sharing a terminal series vertex do
    Assume $c(e) \leq c(f)$;
    $H := H/e$;
    $w := w + c(e)$
  end;
  if $\mathcal{M}$ is the class $B_k$ for some $k > 0$ then
    for each isolated loop $e$ on a terminal vertex do
      $H := H/e$;
      $w := w + c(e)$
  end;
  Delete all isolated terminal vertices from $H$;
  return $(H, w)$
end

Figure 6.4: The Base-Cost burner
prove that Base-Cost correctly computes the cost of an optimum base of $B_k(X(\Gamma))$ or $C_k(X(\Gamma))$. Again, for some $k > 0$, let $\mathcal{M}$ be the class $B_k$ or $C_k$. Let $G$ be any graph.

**Lemma 6.4.1** If $e \in E(G)$ belongs to no circuits of $\mathcal{M}(G)$, then $\{e\}$ is a cocircuit of $\mathcal{M}(G)$.

**Proof:** Since $e$ belongs to no circuits, it belongs to every base. Hence, by Fact 6.1.1, $\{e\}$ is a cocircuit of $\mathcal{M}(G)$. \qed

**Lemma 6.4.2** Let $e, f \in E(G)$ be edges in series such that some circuit of $\mathcal{M}(G)$ contains $e$ or $f$. Then, $\{e, f\}$ is a cocircuit of $\mathcal{M}(G)$.

**Proof:** Let $B$ be any base of $\mathcal{M}(G)$, and suppose $e \notin B$. Consider $B \cup e$. By Fact 6.1.2, $B \cup e$ contains a unique circuit $X$ of $\mathcal{M}(G)$, and $e \in X$. Since $e$ and $f$ are in series, they belong to the same cycles of $G$, and hence, to the same circuits of $\mathcal{M}(G)$. Then, $f \in X - e \subseteq B$. Thus, $e$ or $f$ belongs to every base of $\mathcal{M}(G)$.

Let $X$ be a circuit of $\mathcal{M}(G)$ containing $e$ and $f$. Then, $X - e \in I$ and $X - f \in I$. By axiom $M3$, $X - e \subseteq A$ and $X - f \subseteq B$ for some bases $A$ and $B$ of $\mathcal{M}(G)$. Clearly, $f \notin A$ and $e \notin B$. By Fact 6.1.1, it follows that $\{e, f\}$ is a cocircuit of $\mathcal{M}(G)$. \qed

**Lemma 6.4.3** Every edge $e \in E(G)$ contracted by the burner satisfies $HG2$.

**Proof:** Given the burner, either (1) $e$ is a pendant edge with a terminal pendant vertex, (2) $e$ shares a terminal series vertex with an edge $f$ for which $c(e) \leq c(f)$, or (3) $e$ is an isolated loop on a terminal vertex, and $\mathcal{M}$ is the class $B_k$ for some $k > 0$.

For any $(H, L) \in Env(G)$, the case that applies to $e$ in $G$ also applies to $e$ in $H \circ_L G$. Let $M = \mathcal{M}(H \circ_L G)$. If (1) or (3) applies, then $e$ belongs to no circuits of $M$. By Lemma 6.4.1, $\{e\}$ is a cocircuit of $M$. Suppose (2) applies. If any circuit of
$M$ contains $e$, then \{e, f\} is a cocircuit of $M$ by Lemma 6.4.2. Otherwise, \{e\} is a cocircuit of $M$ by Lemma 6.4.1. In all cases, $M$ has a cocircuit $X \subseteq E(G)$ in which $e$ is of lowest cost. Therefore, by Lemmas 6.2.1 and 6.3.2, $e$ satisfies HG2. □

We can now prove the correctness of algorithm Base-Cost. The proof relies on the correctness of algorithm H-Greedy.

**Theorem 6.4.1** Let $\mathcal{M}$ be the class $C_k$ or $B_k$ for some $k > 0$. Then, applied to hierarchical graph $\Gamma$, algorithm Base-Cost returns the cost of an optimum base of $\mathcal{M}(X(\Gamma))$.

**Proof:** Consider $T^b_\|^\Gamma$, the last pair computed by algorithm Base-Cost. Recall that H-Greedy is nondeterministic. By Lemmas 6.3.1 and 6.4.3, it follows that $T^b_\|^\Gamma.G = H$ and $T^b_\|^\Gamma.w = c(E)$, where $(H, E)$ is the last pair computed by some valid execution sequence of H-Greedy on $\Gamma$. By Theorem 6.2.1, $E(H) \cup E$ is an optimum base of $\mathcal{M}(X(\Gamma))$. Since $E(H)$ and $E$ are disjoint, $c(E(T^b_\|^\Gamma.G)) + T^b_\|^\Gamma.w$ is the cost of an optimum base of $\mathcal{M}(X(\Gamma))$. □

### 6.4.2 Run-time analysis

We analyze algorithm Base-Cost in two parts. First, we show that the burner has an implementation that operates in polynomial time. Next, we show that the burner produces graphs whose sizes are linear in their number of pins. By Theorem 3.1.2 it then follows that Base-Cost operates in time polynomial in the size of $\Gamma$.

Assume the burner is applied to the pair $(G, w)$. STEP 1 finds a subgraph $H$ of $G$ such that $V(H) = V(G)$ and $E(H)$ is an optimum base of the matroid defined on $G$. Optimum bases of the matroids we have considered can be found in polynomial time using the greedy algorithm. Two special cases for which more efficient algorithms
are known are the cycle matroid \((C_1)\) and the bicircular matroid \((B_1)\). A minimum spanning forest can be found in \(O(|V| + |E| \log \log* |E|)\) time \([12]\), while an optimum base for a bicircular matroid, called a \textit{minimum spanning pseudoforest}, can be found in linear time \([11]\). STEP 2 of the burner contracts edges from \(H\). Using depth-first search \([34]\) for each for-loop provides a linear time algorithm for STEP 2. Thus, for each \(k > 0\), polynomial-time burners exist for \(C_k\) and \(B_k\).

We now show that the burner produces graphs whose sizes are linear in their number of pins. The constant of proportionality depends on the integer \(k\) associated with the particular matroid class in question (i.e., \(C_k\) or \(B_k\)).

\textbf{Lemma 6.4.4} Let \(G\) be a connected graph containing \(k\) independent cycles and \(p\) pins. Let \(H\) be the result of applying STEP 2 of the burner for \(C_{k+1}\) to \(G\). Then, \(H\) has at most \(2p + 2k - 2\) vertices and \(2p + 3k - 3\) edges.

\textbf{Proof:}

\subsection*{\(k = 0\):} Then, \(G\) is a tree. Assume \(p > 0\) (otherwise, \(H\) is an empty graph). \(H\) is a tree in which every leaf and series vertex is a pin. Assume \(H\) has \(p_1\) leaves and \(p_2\) series vertices. Let \(H'\) be the result of contracting one edge incident on each series vertex of \(H\). Then, \(|V(H)| = |V(H')| + p_2\) and \(|E(H)| = |E(H')| + p_2\). \(H'\) has \(p_1\) leaves and no series vertices, and hence, has at most \(2p_1 - 2\) vertices and \(2p_1 - 3\) edges. The result follows since \(p \geq p_1 + p_2\).

\subsection*{\(k > 0\):} Then, \(G\) is a tree plus an additional \(k\) non-tree edges. Given the burner, \(H\) is also a connected graph containing \(k\) independent cycles. Every leaf and series vertex of \(H\) is a pin.

Consider \(H - e\) for any \(e \in E(H)\). \(H - e\) has at most (1) two terminal series vertices, or (2) one terminal leaf (if \(e\) was a loop). Let \(J\) be the result of applying STEP 2 of the burner to \(H - e\). In case (1), two vertices and two edges would be
burned off. In case (2), contracting the pendant edge could result in a new terminal series vertex which would also be burned off. In either case, at most two vertices and two edges can be burned off of $H - e$. Therefore, $|V(H)| \leq |V(J)| + 2$ and $|E(H)| \leq |E(J)| + 3$.

Let $e_1, \ldots, e_k$ be any $k$ edges whose deletion from $H$ leaves a tree. Repeat the above process for each of the $k$ edges. In the resulting tree $T$, every leaf and series vertex is a pin, and hence, $T$ has at most $2p - 2$ vertices and $2p - 3$ edges. Therefore, $H$ has at most $2p - 2 + 2k$ vertices and $2p - 3 + 3k$ edges. □

**Lemma 6.4.5** Let $G$ be a graph with $p$ pins, and let $G^b$ be the result of applying to $G$ the burner for $C_k$ (for some $k$). Then, $G^b$ has at most $2p + 2(k - 1) - 2$ vertices and $2p + 3(k - 1) - 3$ edges.

**Proof:** STEP 1 of the burner finds $H \subseteq G$ such that $E(H)$ is a base of $C_k(G)$. Thus, $H$ contains at most $k - 1$ independent cycles. Given the contractions made by the burner, $G^b$ contains the same number of independent cycles as $H$.

Let $C_1, \ldots, C_m$ be the connected components of $G^b$, and assume $C_i$ contains $k_i$ independent cycles. Together, any $C_i$ and $C_j$ contain $k_i + k_j$ independent cycles. Thus, since $G^b$ has at most $k - 1$ independent cycles, $\sum_{i=1}^{m} k_i \leq k - 1$. Assume $C_i$ has $v_i$ vertices, $e_i$ edges, and $p_i$ pins. By Lemma 6.4.4, $v_i \leq 2k_i + 2p_i - 2$, and $e_i \leq 3k_i + 2p_i - 3$. Thus, $|V(G^b)| = \sum_{i=1}^{m} v_i \leq 2(k - 1) + 2p - 2$, and $|E(G^b)| = \sum_{i=1}^{m} e_i \leq 3(k - 1) + 2p - 3$. □

**Lemma 6.4.6** Let $G$ be a connected graph that is not a tree, and let $k \geq 0$. Then, $G$ contains $k + 1$ independent cycles if and only if $G$ contains $k$ independent bicycles.

**Proof:**

"Only If": Suppose $G$ contains $k + 1$ independent cycles. The deletion of $k + 1$ edges
from $G$ is necessary and sufficient to produce a graph with no cycles. Thus, the
deletion of $k$ edges from $G$ is sufficient to produce a graph containing no bicycles.

If $k = 0$, $G$ contains no bicycles. Assume $k > 0$. Let $H$ be the result of deleting
any $k - 1$ edges from $G$. Assume $H$ has $j$ connected components. Then, since $G$ was
connected, it follows that when deleted, $j - 1$ of the edges were bridges (i.e., belonged
to no cycles), and the other $k - j$ were not. Therefore, $H$ contains $(k + 1) - (k - j) =
j + 1$ independent cycles. Since $j > 0$, some connected component contains at least
two cycles, and hence, a bicycle. Therefore, $G$ contains $k$ independent bicycles.

"If": Suppose $G$ contains $k$ independent bicycles. Then, the deletion of any $k$ edges
from $G$ must leave at least one cycle. Thus, $G$ contains at least $k + 1$ independent
cycles. However, by the "Only If" part of this proof, $G$ cannot have more than $k + 1$
independent cycles, for otherwise it has more than $k$ independent bicycles. \hfill $\Box$

Lemma 6.4.7 Let $G$ be a graph with $p$ pins, and let $G^b$ be the result of applying to
$G$ the burner for $B_k$ (for some $k$). Then, $G^b$ has at most $2p + 2(k - 1)$ vertices and
$2p + 3(k - 1)$ edges.

Proof: STEP 1 of the burner $H \subseteq G$ such that $E(H)$ is a base of $B_k(G)$. Thus,$H$
contains at most $k - 1$ independent bicycles. Given the contractions made by the
burner, it follows that $G^b$ contains the same number of independent bicycles as $H$.

Let $C_1, \ldots, C_m$ be the connected components of $G^b$, and assume $C_i$ contains $k_i$
independent bicycles. Together, any $C_i$ and $C_j$ contain $k_i + k_j$ independent bicycles.
Thus, $\sum_{i=1}^m k_i \leq k - 1$. Assume $C_i$ contains $v_i$ vertices, $e_i$ edges, and $p_i$ pins. By
Lemma 6.4.6, $v_i \leq 2(k_i + 1) + 2p_i - 2 = 2k_i + 2p_i$ and $e_i \leq 3(k_i + 1) + 2p_i - 3 = 3k_i + 2p_i$.
Therefore, $|V(G^b)| = \sum_{i=1}^m v_i \leq 2(k - 1) + 2p$, and $|E(G^b)| = \sum_{i=1}^m e_i \leq 3(k - 1) + 2p$.
\hfill $\Box$
Let $G^b$ be the result of applying the burner for some $C_k$ or $B_k$ to a graph $G$ with $p$ pins. By Lemmas 6.4.5 and 6.4.7, $G^b$ has size linear in $k$ and $p$. For a particular burner, $k$ is a constant, so $G^b$ has size $O(p)$. Thus, we have the following theorem.

**Theorem 6.4.2** Applied to hierarchical graph $\Gamma$, algorithm Base-Cost returns the cost of an optimum base of $B_k(X(\Gamma))$ or $C_k(X(\Gamma))$ in time polynomial in the size of $\Gamma$.

### 6.5 Construction Algorithms

In this section, we present a method to generate an optimum base of $C_k(X(\Gamma))$ or $B_k(X(\Gamma))$ using work-space polynomial in the size of $\Gamma$. The method is implemented by an algorithm we call Gen-Base. The burner $B$ for Gen-Base is almost identical to that of algorithm Base-Cost (see Section 6.4) except that it returns the set of edges contracted instead of their cost. That is, $B((G, \emptyset)) = (G^b, C)$, where $G^b$ is as in Section 6.4, and $C$ is the set of edges contracted from $G$.

Let $\Gamma = (G_1, \ldots, G_n)$. Like H-Greedy, Gen-Base manipulates pairs $R$ consisting of a graph $R.G$ and an edge set $R.E$. Gen-Base constructs sequences of pairs $\tilde{T}_1, \ldots, \tilde{T}_n$ and $T^b_1, \ldots, T^b_n$ in which $\tilde{T}_i.E = \emptyset$, and $T^b_i.E$ is the set of edges contracted from $\tilde{T}_i.G$ by the burner. Notice that $T^b_i.G$ is a minor of $X(\Gamma_i)$. Thus, the set of all edges contracted from $X(\Gamma_i)$ to form $T^b_i.G$ is not stored in $T^b_i.E$ (as is done by H-Greedy), but is instead distributed over the hierarchy, thus ensuring that Gen-Base is space-efficient.

Let $\mathcal{M}$ be the matroid class $C_k$ or $B_k$ for some $k > 0$. Gen-Base is somewhat different than the other algorithms we have given, so we present it in Figure 6.5. It utilizes a recursive procedure WriteBase that traverses the hierarchy tree of $\Gamma$ and
writes out the elements of the base. The correctness of Gen-Base follows from the correctness of algorithms Cost-Greedy and H-Greedy.

\[ T_i^b \cdot E \] is the set of edges contracted from \( \bar{T}_i \cdot G \) by the burner. Since \( |\bar{T}_i \cdot G| = O(|G_i|) \), \( T_i^b \cdot E \) has \( O(|G_i|) \) elements. However, each edge of \( \bar{T}_i \cdot G \) is labeled with a pathname in the hierarchy tree of \( \Gamma_i = (G_1, \ldots, G_i) \). Such a path name has at most \( i \) symbols, each requiring \( O(\log n) \) space, where \( n \) is the number of cells of \( \Gamma \). Thus, \( T_i \) and \( T_i^b \) have total size \( O(|G_i| \cdot i \cdot \log n) \). Therefore, the total space used by Gen-Base is \( O(|\Gamma| \cdot n \cdot \log n) \).

Clearly, WriteBase takes time exponential in \( |\Gamma| \) in the worst case, since an optimum base of a matroid on \( X(\Gamma) \) can be as large as \( X(\Gamma) \) itself. However, aside from the time used by WriteBase, algorithm Gen-Base runs in polynomial time.

Note that the sets \( T_i^b \cdot E, \ldots, T_n^b \cdot E \) along with a description of the hierarchy tree of \( \Gamma \) constitute a succinct representation of an optimum base of a matroid on \( X(\Gamma) \). The size of this representation is \( O(|\Gamma| \cdot n \cdot \log n) \).

It is noteworthy to point out that the \( n \cdot \log n \) factor can be removed from the above space bounds by using some of the techniques presented in Chapter 5 for generating forbidden subgraphs. Although these techniques save space by eliminating the need for explicitly storing pathnames, every time an edge is to be written out, a path of pointers must be followed to find its pathname.

### 6.6 Summary and Further Results

We developed H-Greedy, a hierarchical greedy algorithm that generates optimum bases of matroids defined on the expansions of hierarchical graphs. Using H-Greedy, we constructed polynomial-time hierarchical algorithms for computing costs of optimum bases, and polynomial-space hierarchical algorithms for generating optimum
Algorithm Gen-Base

input: $\Gamma = (G_1, \ldots, G_n)$
output: an optimum base of $M(X(\Gamma))$

for $i := 1$ to $n$ do
    if $G_i$ has no nonterminals then
        $\tilde{T}_i := (G_i, \emptyset)$
    else
        Let the nonterminals of $G_i$ be $N_i = \{1*G_{i1}, \ldots, m*G_{im}\}$;
        $\tilde{T}_i := \left( (G_i - N_i) \circ 1*G_{i1}^b \circ \ldots \circ m*G_{im}^b, \emptyset \right)$
    end;
    $T_i^b := B(\tilde{T}_i)$
end;

Output the edges in $E(T_n^b, G)$;
WriteBase(n, $\epsilon$) ( $\epsilon$ is the empty string)

procedure WriteBase(i : integer; $\sigma$ : pathname) begin
    Output the edges in $\sigma * T_i^b * E$;
    for each nonterminal $l*G_j$ of cell $G_i$ do
        WriteBase(j, $\sigma * l$)
    end
end

Figure 6.5: Algorithm Gen-Base
bases of matroids from two infinite families of matroids. The circuits of these ma-
troids constitute homeomorphic matroidal families that each have a finite number of
homeomorphism-minimal graphs. This finiteness appears to be the main reason for
the existence of efficient hierarchical algorithms for these matroids.

H-Greedy can also be applied the head partition and uniform matroids dis-
cussed in Section 6.1. The circuits of these matroids can be characterized by finite,
non-homeomorphic matroidal families. The matroidal family of the head-partition
matroid contains only two graphs: two edges with the same heads but different tails,
and two edges with the same heads and same tails. For the class of uniform matroids
in which any $m$ or fewer elements are independent, the matroidal family contains
all graphs with $m + 1$ edges and no isolated vertices. The simplicity of the circuits
of these matroids leads to efficient optimization algorithms based on H-Greedy. It
would be interesting to know whether there are other classes of problems to which
H-Greedy, or a similar method, may be applied.
7. SUMMARY AND OPEN PROBLEMS

We developed a uniform approach for constructing and proving the correctness of hierarchical algorithms. This approach was used on the following problems:

- Connectivity augmentation problems, where the goal is to compute the minimum number of edges that must be added to a graph to make it satisfy some connectivity property. We provided polynomial-time hierarchical algorithms for bridge-connectivity, biconnectivity, and strong-connectivity augmentation.

- Forbidden subgraph problems, in which one determines if a graph contains a subgraph homeomorphic from a member of some finite set of forbidden graphs. We developed linear-time hierarchical algorithms for the sets \( \{K_4\} \) and \( \{K_4, K_{2,3}\} \), which define, respectively, series-parallel and outer-planar graphs. The techniques used in the algorithms also allowed for listing the edges of a forbidden subgraph of \( X(\Gamma) \), when one exists, using work-space linear in the size of \( \Gamma \).

- Matroid optimization problems, where one determines the cost of an optimum base of a matroid defined on the edge set of a graph. We developed polynomial-time hierarchical algorithms for two infinite families of matroid classes whose circuits can be characterized by sets of graphs closed under homeomorphism. In addition, we developed polynomial-space hierarchical algorithms for generating optimum bases instead of their costs.
Our approach came about from abstractions of graph gluing, the bottom-up method, and replaceability. Each algorithm relied on a burner function to "shrink" the graph or tuple it was applied to. Although the burners seem quite different, one technique was common to all of them: construction of minors. Each burner employed edge deletion, vertex deletion, and edge contraction to reduce the size of the graph being burnt (although the augmentation algorithms applied these operations to bc-forests, br-forests, and st-dags). By maintaining the parts of the graph which were contracted or deleted, in some cases we were able to generate subgraphs of $X(\Gamma)$ using space polynomial in the size of $\Gamma$.

There are many open problems and unanswered questions concerning hierarchical graphs. Compared to the number of non-hierarchical graph algorithms, the number of hierarchical algorithms is quite small. We have examined only a few problems in detail, and there could be, and probably are, many other problems for which efficient hierarchical algorithms exist.

While finding new hierarchical algorithms is attractive, it is equally important to determine properties of graph problems sufficient to ensure polynomial-time hierarchical complexity. One property seems to be that only "local" information is needed to solve the problem on non-hierarchical graphs. In an attempt to solve this problem, one must formalize these notions, a task which in itself seems quite formidable.

Lengauer and Wagner [21] considered the hierarchical complexity of many problems. They found, for example, that the Circuit Value Problem (CVP) is PSPACE-complete under the hierarchical model. Clearly, CVP is important to determining the validity of circuit designs. Thus, even though CVP is PSPACE-complete, it is important to attempt to find a hierarchical algorithm for CVP that is as time or space-efficient as is possible. Such an algorithm may be faster in many cases than
expanding the hierarchy and working directly with the expansion.

There are other issues of practical importance concerning hierarchical algorithms. The algorithms presented here and those given in [19,20,22,24] are asymptotically efficient. It would be interesting to compare the running time of a hierarchical algorithm to that of simply constructing $X(\Gamma)$ and applying a non-hierarchical algorithm. The non-hierarchical approach will almost certainly be faster when the expansion of $\Gamma$ is not much larger than $\Gamma$. What relationship must exist between the sizes of $\Gamma$ and $X(\Gamma)$ to ensure that the hierarchical algorithm runs faster than a non-hierarchical one? Are there any "real" problem instances that satisfy this relationship? In their favor, hierarchical algorithms may use substantially less space than required for the non-hierarchical approach, thus saving on virtual memory overhead.
8. BIBLIOGRAPHY


