UNIVERSITÀ DEGLI STUDI ROMA TRE

Department of Civil, Computer Science and Aeronautical Technologies Engineering

Master's Degree Course in Computer Engineering

# Enumeration of 1-Page Book Embeddings of Directed Acyclic Graphs

By

**Ivan Carlini**

Supervisor

**Prof. Giordano Da Lozzo**

Academic Year 2023/2024

*To those who are close to me.*

# Acknowledgments

Il lavoro che segue racchiude sei mesi della mia vita. Sei mesi durante i quali mi sono posto l'obiettivo di concludere al meglio quelli che sono stati cinque anni di studi impegnativi ma anche molto affascinanti. La fortuna ha voluto che gran parte di ciò che ha accopagnato questi anni sono state esperienze positive e questo mi ha consentito di affrontare gli studi con la dovuta serenità e con lo spirito giusto. Vorrei quindi utilizzare queste righe per ringraziare innanzitutto il mio relatore, il Professor Giordano Da Lozzo, per avermi guidato e supportato, e soprattutto per avermi dato l'opportunità di lavorare su un progetto che rispecchia pienamente i miei interessi. Ringrazio inoltre la mia famiglia, che mi ha sostenuto durante il percorso di studi, e i miei amici e colleghi, che mi sono stati accanto durante questa fase universitaria della mia vita.

# Contents

**Contents** **iv**

**List of Figures** **vi**

**List of Algorithms** **viii**

**Abstract** **ix**

**Introduction** **x**

**1 Preliminaries** **1**

1.1 Undirected and Directed Graphs . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Block-Cutpoint Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2.1 BFS Ordering of the Blocks in a Block-Cutpoint Tree . . . . . . 4

1.3 Graph Drawing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3.1 Planar Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3.2 Outerplanar Graphs . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.4 Book Embeddings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.5 Enumeration Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**2 The Algorithm for Recognizing 1-Stack Layouts by Heath and Pemmaraju** **9**

2.1 The Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.1.1 First Step of the Algorithm . . . . . . . . . . . . . . . . . . . . . 10

2.1.2 Second Step of the Algorithm . . . . . . . . . . . . . . . . . . . . 12

2.2 Correction of an Inaccuracy in the Algorithm . . . . . 19

2.3 Towards an Enumeration Algorithm . . . . . 22

**3 A Linear-Delay Algorithm for Enumerating 1-Stack Layouts 23**

3.1 The Algorithm . . . . . 23

3.1.1 Labeling of the Edges of the Block-Cutpoint Tree . . . . . 24

3.1.2 Finding a Better Initial Rooting . . . . . 25

3.1.3 Restricted Paths and Permutations of Blocks . . . . . 26

3.1.4 The Enumeration Process . . . . . 30

3.1.5 The Final Algorithm . . . . . 34

3.2 Counting the Number of 1-Stack Layouts . . . . . 34

3.3 Correctness of the Algorithm . . . . . 37

**4 A Web System for Enumerating 1-Stack Layouts 43**

4.1 Implementation of the Algorithm . . . . . 43

4.2 WebAssembly and Web Interface . . . . . 45

4.2.1 Encountered Challenges . . . . . 46

4.3 API and Interface . . . . . 47

**Conclusions and Future Developments 48**

Theoretical Future Developments . . . . . 48

Practical Future Developments . . . . . 49

**Bibliography 50**

# List of Figures

1.1 A directed acyclic graph (DAG) with its biconnected components circled. The Hamiltonian paths within each biconnected component are highlighted with different colors. . . . 2

1.2 Block-cutpoint trees of the DAG in Figure 1.1. . . . 4

1.3 An outerplanar graph. . . . 6

1.4 A 1-page book embedding of the DAG in Figure 1.1. . . . 7

2.1 Example directed acyclic graph (DAG). . . . 9

2.2 Example of a biconnected outerplanar DAG $\vec{B}$ that does not contain a directed Hamiltonian path traversing the outer face of an outerplanar embedding of $\vec{B}$. . . . 11

2.3 Biconnected components of the DAG in Figure 2.1. The highlighted vertices are intermediate vertices within a biconnected component. Specifically, vertex 1 is intermediate in $\vec{B}_6$, and vertex 6 is intermediate in $\vec{B}_3$. . . . 11

2.4 1-stack layouts of the biconnected components of the DAG in Figure 2.3. . . 12

2.5 Block-cutpoint tree of the DAG in Figure 2.3. . . . 13

2.6 Highlighted on the left block-cutpoint tree are the nodes belonging to $D(\vec{B}_4, 1)$, while highlighted on the right block-cutpoint tree are the nodes belonging to $D(\vec{B}_3, 3)$. . . . 14

2.7 Example of a DAG with a conflicting pair of cutpoints. . . . 14

2.8 Vertices highlighted in white are exposed in the topological order shown, indicating that no arcs exist where the vertex lies between two other vertices in this order. . . . 17

2.9 Construction of a 1-stack layout for the example DAG depicted in Figure 2.3 using the MERGE1STACKLAYOUTS procedure described in Algorithm 2. For each iteration, the vertices and edges of the biconnected component being added to the current layout are highlighted in red. . . . . . . . . . . . . . . 18
2.10 Example DAG with a conflicting pair of cutpoints. . . . . . . . . . . . . . . 20
2.11 Block-cutpoint trees of the DAG in Figure 2.10 for which the preprocessing procedure returns a correct result. . . . . . . . . . . . . . . . . . . . . . . . 21
2.12 Block-cutpoint trees of the DAG in Figure 2.10 for which the preprocessing procedure returns an incorrect result. . . . . . . . . . . . . . . . . . . . . . . . 21

3.1 The first and last cutpoints appearing on the highlighted path, denoted as $u$ and $v$, constitute a conflicting pair of cutpoints. . . . . . . . . . . . . . . . 24
3.2 Labeled block-cutpoint tree of the DAG in Figure 2.3. . . . . . . . . . . . . 25
3.3 Block-cutpoint tree $T_6$ obtained after re-rooting the block-cutpoint tree at block $\vec{B}_6$, following the identification of 6 as the *betterRoot* during the COMPUTERESTRICTIONS procedure. This re-rooting eliminates any cutpoint restrictions, simplifying subsequent computations. . . . . . . . . . . . . . . . . 27
3.4 Highlighted in red are four distinct restricted paths in the block-cutpoint tree depicted in Figure 3.3. Note that all restricted paths are directed towards the root, as this block-cutpoint tree is rooted at block $\vec{B}_{betterRoot}$. . . . . . . 28
3.5 Highlighted in green are the permutable blocks identified in the example block-cutpoint tree $T_{betterRoot}$. . . . . . . . . . . . . . . . . . . . . . . . . 29
3.6 Highlighted in green are the child blocks of a cutpoint that are permutable in the two rootings obtained for the example DAG. . . . . . . . . . . . . . 36

4.1 The web interface for enumerating 1-stack layouts. . . . . . . . . . . . . . . 46

# List of Algorithms

1 Preprocessing Step . . . . . . 17
2 Merge 1-Stack Layouts . . . . . . 19
3 Preprocessing Step (Updated) . . . . . . 22
4 Compute Restrictions . . . . . . 26
5 Find Other Roots . . . . . . 30
6 BCTPermutationEnumerator Struct Definition . . . . . . 31
7 Initialize Enumerator . . . . . . 32
8 Get Next Permutation . . . . . . 33
9 Enumerate 1-Stack Layouts . . . . . . 35

# Abstract

In this thesis, we address the problem of enumerating 1-page book embeddings, also known as 1-stack layouts, of directed acyclic graphs (DAGs). We first introduce and discuss an existing linear-time algorithm developed by Heath and Pemmaraju more than two decades ago, which determines whether a given DAG admits a 1-stack layout and, in the positive case, outputs one such layout. Building on their work, we devised a novel algorithm that enumerates all possible 1-stack layouts of a given DAG without repetitions. The proposed algorithm achieves a time delay of $O(n)$ between consecutive layouts, where $n$ represents the number of nodes in the graph. It leverages the structure of the block-cutpoint tree to efficiently handle cutpoints and permutable blocks, allowing for the systematic generation of distinct layouts. Additionally, we derive a formula to count the total number of possible 1-stack layouts for a given DAG, thus contributing to the theoretical understanding of the problem. Furthermore, the algorithm has been implemented as a C++ library adopted in a WebAssembly application, enabling an efficient, web-based interface that supports interactive exploration of the generated layouts.

# Introduction

Graph Drawing and layout problems play a fundamental role in many areas of Computer Science, including Information Visualization, Network Analysis, and Parallel Computing. In particular, directed acyclic graphs (DAGs) are widely used to model processes with dependencies, such as scheduling tasks, circuit design, and data flow analysis. Representing these graphs in an efficient and visually clear manner is crucial for understanding their structure and extracting useful insights. One approach to achieving this is through book embeddings, where the vertices of the graph are arranged along a line (referred to as the spine), and edges are assigned to different pages, ensuring no edges within a page cross.

The primary focus of this thesis are the 1-page book embeddings of DAGs, also known as 1-stack layouts. In such embeddings, all edges must be drawn on a single page, and the vertices must follow a topological order, reflecting the acyclic nature of the graph. The problem of determining whether a DAG can be embedded in a single page has been addressed by Heath and Pemmaraju [HP99], who developed an algorithm to recognize 1-stack layouts in linear time. However, this thesis goes beyond recognition and proposes a new algorithm to enumerate all possible 1-stack layouts for a given DAG.

Graph layout algorithms are essential for creating readable visual representations of graphs, especially when working with large datasets. For example, layouts that minimize crossings or optimize edge placement can significantly improve clarity. Traditional methods for calculating graph layouts, however, can be computationally expensive, especially for large graphs with complex structures. To address this challenge, linear-time algorithms have been developed to compute certain graph layouts efficiently. A linear-time algorithm is particularly valuable in scenarios where the input size (in this case, the number of vertices and edges) is large, and performance is a critical factor.

The existing algorithm by Heath and Pemmaraju recognizes whether a DAG can be embedded in a single page by decomposing the graph into biconnected components, ensuring each component is 1-page book embeddable, and then combining, if possible, the layouts of the components. This approach runs in $O(n)$ time, where $n$ is the number of vertices in the graph, making it highly efficient for practical applications.

Building on this foundation, the goal of this thesis is to introduce an algorithm that not only recognizes graphs that admit 1-stack layouts, but enumerates all distinct 1-stack layouts of a DAG. The ability to enumerate all possible layouts opens the door to further exploration of layout optimization, as different layouts may offer different advantages depending on the specific application or visualization requirement. The enumeration algorithm presented in this thesis achieves a time delay of $O(n)$ between consecutive layouts, meaning that each new layout can be generated in linear time relative to the number of vertices in the graph.

The algorithm leverages the structure of the block-cutpoint tree to manage the graph's biconnected components efficiently, and handles cutpoints and permutable blocks to systematically generate all possible 1-stack layouts. By suitably re-rooting the block-cutpoint tree and permuting certain blocks, the algorithm ensures that all valid layouts are explored without redundancy.

In addition to the theoretical contributions, this thesis also includes a practical component: The implementation of the enumeration algorithm in WebAssembly. This implementation allows users to interact with the algorithm through a web interface, providing a tool for visualizing and exploring the different layouts of a given DAG. The system demonstrates the efficiency and applicability of the algorithm in real-world scenarios, offering a platform for further research and development in graph layout techniques.

Overall, this thesis contributes to the field of Graph Drawing by extending existing results on 1-stack layouts of DAGs and by providing a practical tool for graph layout enumeration. The methods and algorithms developed herein are applicable not only to theoretical problems but also to practical challenges in fields such as Data Visualization, Software Engineering, and Operations Research.

The thesis begins by establishing the fundamental concepts in Chapter 1. These include undirected and directed graphs, the block-cutpoint tree structure, Graph Drawing techniques, book embeddings, and enumeration algorithms. These theoretical foundations are crucial for understanding the methods and algorithms developed later in the thesis.

In Chapter 2, the algorithm by Heath and Pemmaraju is presented, which determines whether a directed acyclic graph (DAG) admits a 1-page book embedding, referred to as a 1-stack layout. This linear-time algorithm forms the basis for the enumeration process introduced in the following chapters.

Building on this, Chapter 3 introduces a linear delay algorithm that enumerates all possible 1-stack layouts of a DAG. The algorithm leverages the block-cutpoint tree structure to efficiently manage cutpoints and permutable blocks, ensuring that each layout is generated with a linear time delay between successive results, without redundancy. Additionally, this chapter presents the formula for counting the number of possible 1-page book embeddings of a DAG.

In Chapter 4, the thesis describes the implementation of this enumeration algorithm in a web-based system. The algorithm is compiled into WebAssembly and made accessible via an interactive web interface. This chapter also discusses the challenges encountered during development, along with the API features, demonstrating the practical applicability of the algorithm in real-world scenarios.

The thesis concludes by summarizing the contributions made, both theoretical and practical, and suggests directions for future research and potential improvements to the system.

# Chapter 1

# Preliminaries

This chapter presents several fundamental concepts that form the basis of the main results of this thesis. These include basic definitions and properties of directed and undirected graphs, the block-cutpoint tree structure, Graph Drawing techniques, book embeddings, and enumeration algorithms. Each section provides the theoretical foundation and terminology necessary for the subsequent development and discussion of 1-stack layout enumeration in directed acyclic graphs (DAGs).

## 1.1 Undirected and Directed Graphs

A *graph* $G = (V, E)$ consists of a set of vertices $V$ and a set of edges $E$, where each edge is an unordered pair of vertices. In an *undirected graph*, the edges have no direction, meaning that if there is an edge between two vertices $u$ and $v$, one can traverse the edge in both directions, expressed as $\{u, v\} = \{v, u\}$. Conversely, A *directed graph*, $\vec{G} = (V, \vec{E})$, consists of vertices and directed edges, where each edge is represented as an ordered pair $(u, v)$, indicating a direction from vertex $u$ to vertex $v$.

In the context of directed graphs, a vertex $u$ is termed a *source* if it has no incoming edges, which can be formally stated as $(u, v) \notin \vec{E}$ for any $v \in V$. In contrast, a vertex $v$ is called a *sink* if it has no outgoing edges, expressed as $(u, v) \notin \vec{E}$ for any $u \in V$.

A fundamental concept relevant to graph layouts is the existence of *Hamiltonian paths*. A Hamiltonian path in a graph is a path that visits each vertex exactly once. In undirected graphs, the search for Hamiltonian paths can be particularly challeng-

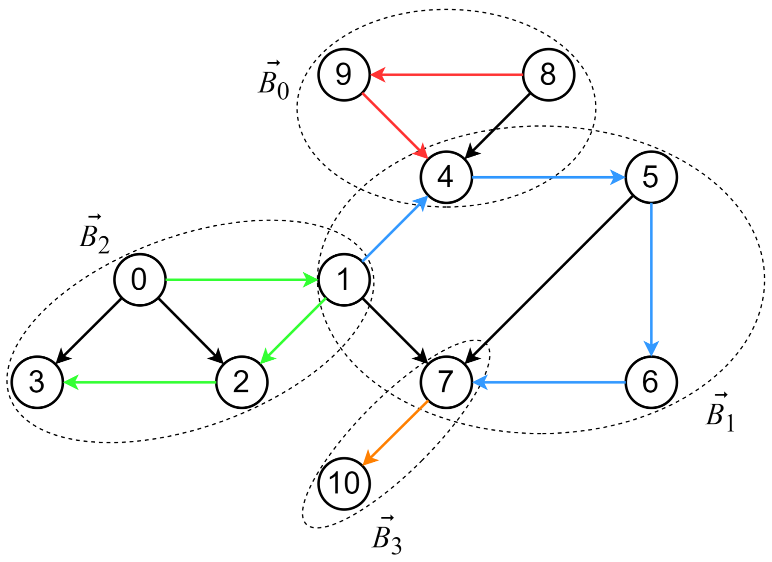

Figure 1.1: A directed acyclic graph (DAG) with its biconnected components circled. The Hamiltonian paths within each biconnected component are highlighted with different colors.

ing; however, certain classes of graphs, such as *outerplanar graphs*, frequently contain Hamiltonian paths, which are critical in determining various embeddings, including *book embeddings* that will be discussed in subsequent sections. In directed graphs, a Hamiltonian path similarly visits each vertex exactly once while adhering to the direction of the edges.

This thesis focuses particularly on *directed acyclic graphs* (DAGs), which are directed graphs that do not contain directed cycles, as shown in Figure 1.1. In a DAG, there exists a *topological ordering* of the vertices, which is a linear ordering such that, for every directed edge $(u, v)$, vertex $u$ comes before vertex $v$ in the ordering [Wes00].

## 1.2 Block-Cutpoint Tree

In Graph Theory, a *block-cutpoint tree* (BCT) is a structure that captures the decomposition of a graph into its *biconnected components* (or blocks) and *cutpoints* (or articulation points). This decomposition is useful in analyzing the connectivity and structural properties of graphs, especially for tasks such as graph drawing and layout generation.

A biconnected component of a graph is defined as a maximal subgraph in which any pair of vertices remains connected after the removal of any single vertex. In other words, a biconnected component does not contain any cutpoints; the removal of any vertex does not disrupt the connectivity of the component. Understanding biconnected components is crucial, as they are often treated as fundamental units in various graph algorithms to facilitate the analysis of the graph's structure.

A cutpoint (or articulation point) is a vertex whose removal increases the number of connected components in the graph. The significance of cutpoints lies in their role as connectors between different biconnected components of the graph.

The block-cutpoint tree of a graph $G$ encapsulates the relationships between its biconnected components and cutpoints. Each node in the block-cutpoint tree corresponds either to a block or to a cutpoint of the graph. Edges in the block-cutpoint tree connect blocks and cutpoints. Specifically, every block is connected to the cutpoints that belong to it and no two blocks are directly connected in the block-cutpoint tree, as they must share a cutpoint to be connected. This decomposition can be efficiently computed using the algorithm developed by Hopcroft and Tarjan [HT73a].

For a connected undirected graph $G$, the block-cutpoint tree provides a decomposition into smaller and simpler components that can be handled individually. This decomposition is particularly useful when working with embeddings and layouts, where the arrangement of vertices within each biconnected component must adhere to specific constraints to achieve certain properties, such as planarity or the existence of 1-stack layouts.

When working with directed acyclic graphs (DAGs), the block-cutpoint tree is similarly constructed based on the underlying undirected version of the graph, ignoring edge directions. This process ensures that the relevant structural characteristics of the DAG are preserved, enabling efficient manipulation and analysis of its components. An example of a block-cutpoint tree is depicted in Figure 1.2.

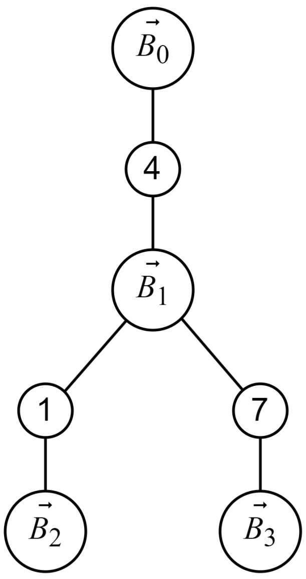

Figure 1.2: Block-cutpoint trees of the DAG in Figure 1.1.

### 1.2.1 BFS Ordering of the Blocks in a Block-Cutpoint Tree

Let $T(\vec{G})$ be a block-cutpoint tree associated with a graph $\vec{G}$, where the nodes of $T(\vec{G})$ correspond to the blocks $\vec{B}_0$, $\vec{B}_1$, ..., $\vec{B}_{m-1}$ of $\vec{G}$ and the cutpoints $C_0$, $C_1$, ..., $C_{k-1}$, and each cutpoint is connected to the blocks it belongs to.

Let $\sigma = \{\vec{B}_{[1]}, \vec{B}_{[2]}, ..., \vec{B}_{[m]}\}$ be an ordering of the blocks of $T(\vec{G})$. The ordering $\sigma$ is a *breadth first search* (BFS) ordering of the blocks of the block-cutpoint tree $T(\vec{G})$ if and only if it satisfies the following conditions:

- There exists a chosen root $\vec{B}_r$ (any block of the BCT) from which the BFS traversal starts. Without loss of generality, $\vec{B}_{[1]} = \vec{B}_r$.
- Blocks from subsequent levels in the BFS traversal can only appear after all blocks from the previous levels have been included in the ordering.
- If multiple blocks are connected to the same cutpoint $C$, they must appear as a contiguous group in the ordering $\sigma$. Specifically, if $\vec{B}_{[i]}$ and $\vec{B}_{[j]}$ are children of the same cutpoint $C$, no block that is not a child of $C$ can appear between $\vec{B}_{[i]}$ and $\vec{B}_{[j]}$ in the ordering.

## 1.3 Graph Drawing

Graph Drawing is a field in Graph Theory that focuses on the visual representation of graphs in a way that conveys their structure clearly and efficiently [DETT99]. The primary objective is to develop drawings that minimize edge crossings, optimize space utilization, and enhance readability, which is crucial in various applications such as network visualization, circuit design, and information representation. This section emphasizes the characteristics of planar and outerplanar graphs, which are particularly relevant for the study of 1-stack layouts.

### 1.3.1 Planar Graphs

A graph is called *planar* if it can be drawn on a plane without any of its edges crossing. This means that the edges of $G$ intersect only at their endpoints, which correspond to the graph's vertices. Planar graphs are fundamental in Graph Drawing because their embeddings can often lead to more readable and understandable layouts, especially in practical applications like circuit design and network visualization.

A classic result about planar graphs is the *Kuratowski's theorem* [KW01], which states that a graph is planar if and only if it does not contain a subgraph that is a subdivision of the complete graph $K_5$ (a complete graph on five vertices) or the complete bipartite graph $K_{3,3}$ (a complete bipartite graph on two sets of three vertices). These graphs are the minimal non-planar graphs and serve as obstructions to planarity.

### 1.3.2 Outerplanar Graphs

A graph is called *outerplanar* if it can be drawn in the plane without edge crossings in such a way that all of its vertices lie on the *outer face* of the drawing. The outer face is defined as the unbounded region surrounding the graph in the plane. An example of an outerplanar graph is shown in Figure 1.3. Outerplanar graphs represent a subclass of planar graphs [KW01], and their structural properties make them particularly suitable for specific layout algorithms, including 1-page book embeddings.

A graph is outerplanar if and only if it does not contain a subdivision of $K_4$ (the complete graph on four vertices) or $K_{2,3}$ (a complete bipartite graph between two vertices

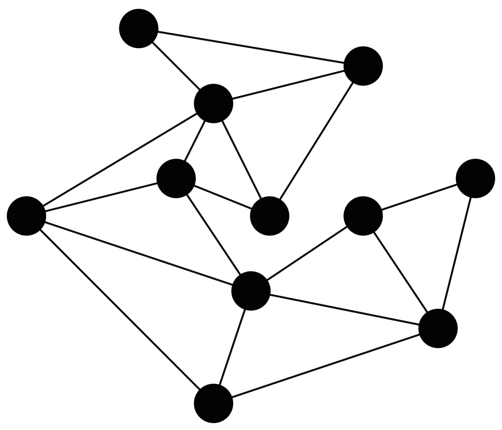

Figure 1.3: An outerplanar graph.

and three vertices). These subgraphs act as obstructions to outerplanarity, analogous to the role of $K_5$ and $K_{3,3}$ in planarity.

## 1.4 Book Embeddings

Book embeddings represent a specific category of graph drawing in which the vertices of a graph are arranged along a spine (a straight line), and the edges are assigned to pages in such a manner that edges on the same page do not cross. Formally, a book embedding of a graph $G$ consists of a linear ordering $\prec$ of the vertices along a line, referred to as the *spine* of the book, and an assignment of the edges to various half-planes delimited by the spine, known as pages. The critical constraint is that no two edges $(u, v)$ and $(w, z)$ on the same page can cross. This is mathematically expressed such that if $u \prec v$ and $v \prec w$, then it must not be the case that $u \prec w \prec v \prec z$. Furthermore, the edges $(u, v)$ and $(w, z)$ nest on the same page if $u \prec w \prec z \prec v$. The *book thickness* of $G$ is defined as the minimum integer $k$ such that $G$ has a book embedding on $k$ pages [BDG+23]. Book embeddings have important applications in areas like VLSI design, sorting networks, and visualization of hierarchical structures.

A 1-page book embedding, also known as 1-stack layouts, arrange vertices along a line with all edges drawn as semi-circles above or below this line, ensuring that no two edges intersect. Notably, any outerplanar undirected graph can achieve a 1-page book

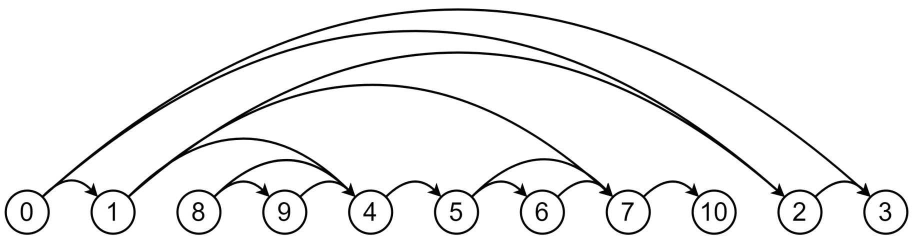

Figure 1.4: A 1-page book embedding of the DAG in Figure 1.1.

embedding, meaning an undirected graph can be represented this way if and only if it is outerplanar.

In the context of DAGs, a 1-stack layout is characterized by a linear ordering of the vertices such that all directed edges can be represented as non-crossing arcs on a single page, adhering to the edges' directionality. This property introduces additional complexity compared to undirected outerplanar graphs. An example of a 1-page book embedding of a DAG is depicted in Figure 1.4. In 1999, Heath and Pemmaraju [HP99] presented a linear time algorithm to test the existence of a 1-stack layout of an input DAG (see also Chapter 2) and showed that testing the existence of a 6-stack layout of a DAG is an NP-complete problem. More recently, it has been shown that the problem of determining the existence of $k$-stack layout for a DAG is NP-complete already for $k = 2$ [BDF+23].

While 1-page book embeddings are known to exist for outerplanar graphs, extending this concept to general graphs requires multiple pages. A graph that admits a book embedding with $k$ pages is said to have a book thickness (or *stack number*) of $k$. For planar graphs, it has been demonstrated that their book embeddings use a maximum of four pages [BK79]. In contrast, the determination of the book thickness for general graphs may require more pages and remains a challenging problem.

## 1.5 Enumeration Algorithms

Enumeration algorithms systematically generate all possible solutions [HP73] or configurations for a given problem. In Graph Drawing, they are particularly important for listing all feasible graph layouts, embeddings, or orderings that meet certain criteria.

This approach is central to combinatorial problems, where one seeks to explore the entire solution space rather than identifying just one optimal solution. Knuth [Knu06] highlights the importance of such methods in combinatorial algorithms, emphasizing their utility in both theoretical investigations and practical applications such as network design and optimization.

In this thesis, the focus is on the enumeration of 1-stack layouts for DAGs. The problem of enumerating all possible 1-stack layouts is a specific case of layout enumeration [DDF+24]. Enumeration in this context involves generating all valid vertex orderings, where the directed edges of the DAG can be drawn as non-crossing arcs on a single page.

A critical aspect of enumeration algorithms is the *time delay* between the generation of successive solutions. Time delay refers to the computational time required to produce the next solution once the current one has been output. In many practical applications, it is important that this delay remains bounded and ideally linear in the size of the input. For instance, when generating permutations or subsets, a delay of $O(n)$, where $n$ is the number of elements, is desirable to ensure efficient generation. The algorithm presented in Chapter 3 achieves a worst-case delay of $O(n)$, where $n$ represents the number of nodes in the graph, between successive 1-stack layouts of a given DAG, making it highly efficient for large graphs.

Another important consideration in enumeration algorithms is *counting*, which refers to determining the total number of valid solutions without explicitly generating them. Counting provides valuable insight into the complexity of a problem and helps estimate the resources required for enumeration. In this thesis, the counting of all possible 1-stack layouts for a given DAG is also addressed, complementing the enumeration process. The method used for counting is discussed in detail in Section 3.2.

Chapter 2

# The Algorithm for Recognizing 1-Stack Layouts by Heath and Pemmaraju

In this chapter, an $O(|V|)$ time algorithm for determining whether a DAG $\vec{G} = (V, \vec{E})$ is a 1-stack DAG presented by Heath and Pemmaraju [HP99] will be described. If $\vec{G}$ is a 1-stack DAG, the algorithm constructs a 1-stack layout of $\vec{G}$. The example DAG shown in Figure 2.1 is used to illustrate the application of the various stages of the algorithm.

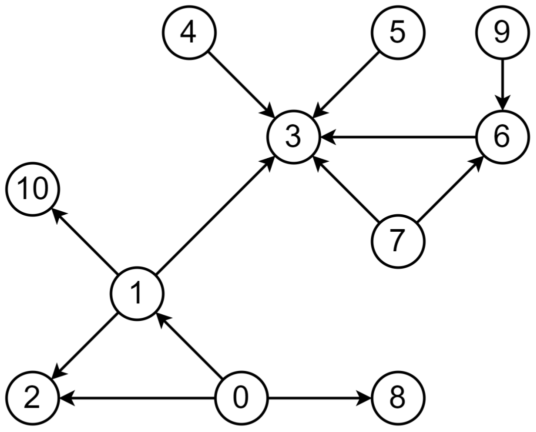

Figure 2.1: Example directed acyclic graph (DAG).

## 2.1 The Algorithm

A DAG is a 1-stack DAG if and only if each of its connected components is a 1-stack DAG. Therefore, it can be assumed, without loss of generality, that $\vec{G}$ is connected. Additionally, for $\vec{G}$ to be a 1-stack DAG, its covering graph $G = (V, E)$ must be a 1-stack graph. According to Bernhart and Kainen's characterization [BK79], this implies that $G$ must be outerplanar, meaning $|E| \leq 2|V| - 3$. Since $|\vec{E}| = |E|$, it can be assumed, without loss of generality, that $|\vec{E}| \leq 2|V| - 3$.

Bernhart and Kainen show that the stack number of an undirected graph is determined by the maximum stack number of its biconnected components. However, this result does not apply to DAGs. Consequently, to verify that a DAG is a 1-stack DAG, it is not sufficient to ensure that each biconnected component is a 1-stack DAG.

The algorithm for recognizing 1-stack DAGs is divided into two main steps. In the first step, it checks whether each biconnected component of $\vec{G}$ is a 1-stack DAG. If any biconnected component is found not to be a 1-stack DAG, the algorithm terminates immediately and reports a failure. In the second step, the algorithm combines the 1-stack layouts of the biconnected components to form a complete 1-stack layout of $\vec{G}$.

### 2.1.1 First Step of the Algorithm

The first step of the algorithm involves decomposing $\vec{G}$ into its biconnected components and verifying that each component is a 1-stack DAG. This verification relies on the following lemma.

**Lemma 1** (Lemma 1.1 in [HP99]). *A biconnected DAG $\vec{B} = (V, \vec{E})$ is a 1-stack DAG if and only if $\vec{B}$ is an outerplanar DAG and contains a directed Hamiltonian path obtained by traversing the outer face of an outerplanar embedding of $\vec{B}$.*

A depth-first search on $\vec{G}$ can be utilized to decompose $\vec{G}$ into its biconnected components [HT73b]. This procedure incurs a time complexity of $O(|V|)$, as the depth-first search on $G$ requires $O(|V| + |E|)$ time, with $|E| \leq 2|V| - 3$. For each biconnected component $\vec{B}$ of $\vec{G}$, one can integrate a topological sort with a verification process to ascertain that $\vec{B}$ contains a directed Hamiltonian path and that the resulting topological

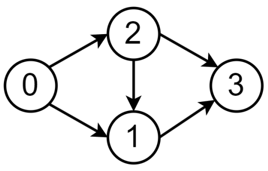

Figure 2.2: Example of a biconnected outerplanar DAG $\vec{B}$ that does not contain a directed Hamiltonian path traversing the outer face of an outerplanar embedding of $\vec{B}$.

order produces a 1-stack layout of $\vec{B}$. The time complexity associated with this initial phase is $O(|V|)$.

Figure 2.2 illustrates an example of a biconnected and outerplanar DAG in which no Hamiltonian path exists along its outer face, while Figure 2.3 presents the decomposition of the example DAG into its biconnected components. Figure 2.4 depicts the 1-stack layouts obtained for each biconnected component of the example DAG.

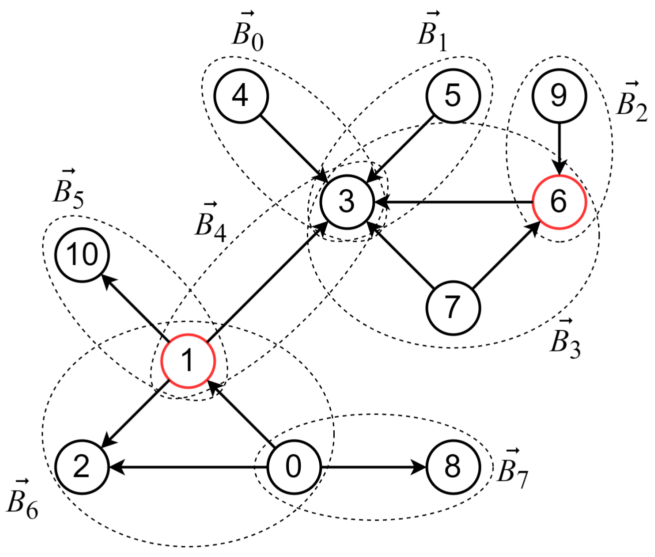

Figure 2.3: Biconnected components of the DAG in Figure 2.1. The highlighted vertices are intermediate vertices within a biconnected component. Specifically, vertex 1 is intermediate in $\vec{B}_6$, and vertex 6 is intermediate in $\vec{B}_3$.

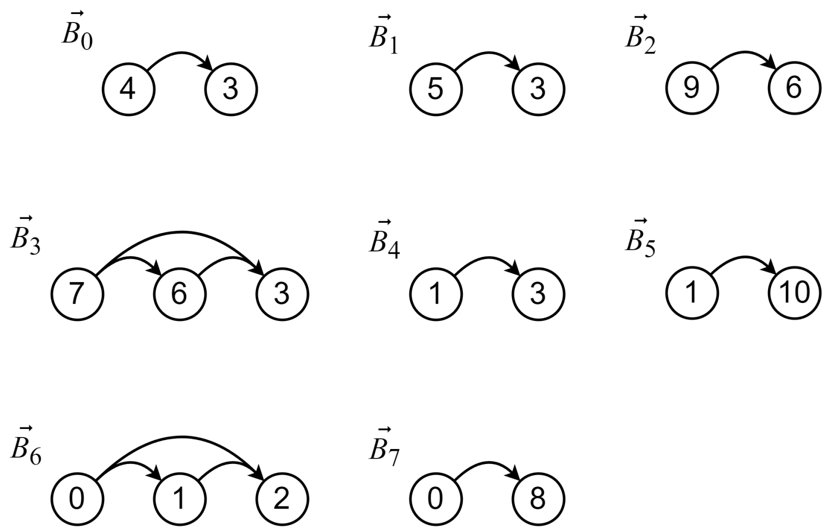

Figure 2.4: 1-stack layouts of the biconnected components of the DAG in Figure 2.3.

### 2.1.2 Second Step of the Algorithm

The second step of the algorithm "merges" the 1-stack layouts of the biconnected components of $\vec{G}$ into a single 1-stack layout of $\vec{G}$. According to Lemma 1, each 1-stack biconnected component of $\vec{G}$ has a unique source (a vertex with in-degree 0), a unique sink (a vertex with out-degree 0), and a unique 1-stack layout. For this step, the concept of a block-cutpoint tree, as defined by Harary and Prins [HP66], is utilized. The block-cutpoint tree $T(\vec{G})$ of a DAG $\vec{G}$ is the undirected graph with node set

$$\{\vec{B} \mid \vec{B} \textit{ is a biconnected component of } \vec{G}\} \cup \{u \mid u \textit{ is a cutpoint of } \vec{G}\}$$

and edge set

$$\{\{\vec{B}, u\} \mid u \textit{ is a cutpoint in } \vec{B}\}$$

Figure 2.5 illustrates the block-cutpoint tree of the DAG shown in Figure 2.3. The larger circles represent the graph's biconnected components, while the smaller circles represent the graph's cutpoints. Harary and Prins show that $T(\vec{G})$ forms a tree if and only if $G$ is connected.

An *intermediate vertex* within a biconnected component $\vec{B}$ is defined as a vertex in $\vec{B}$ that is neither the source nor the sink of $\vec{B}$. For instance, in Figure 2.3, vertex 1 is

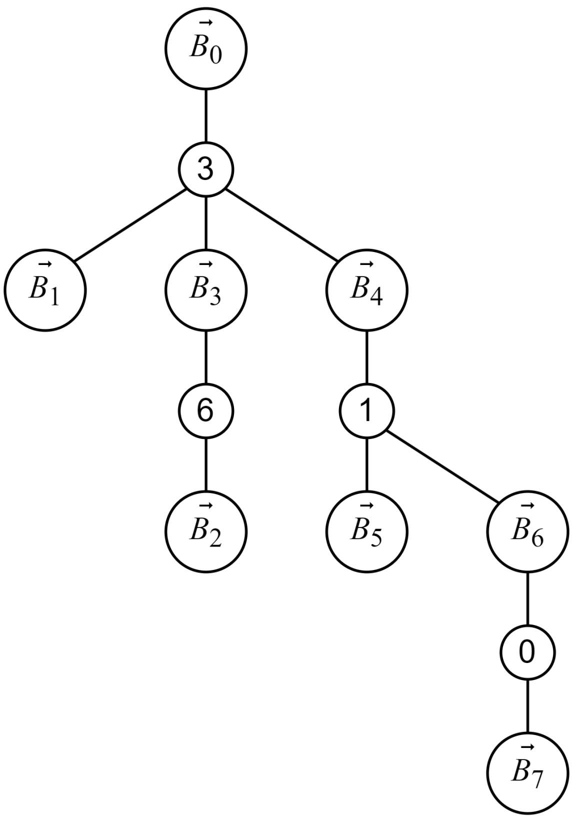

Figure 2.5: Block-cutpoint tree of the DAG in Figure 2.3.

an intermediate vertex in $\vec{B}_6$, and vertex 6 is an intermediate vertex in $\vec{B}_3$. For any biconnected component $\vec{B}_i$ of $\vec{G}$, $T_i$ denotes the tree $T(\vec{G})$ rooted at $\vec{B}_i$. If $x$ is a vertex in $\vec{B}_i$ that also is a cutpoint of $\vec{G}$, $D(\vec{B}_i, x)$ represents the set of nodes in the subtree of $T_i$ rooted at $x$. For example, in the DAG and its corresponding block-cutpoint tree depicted in Figure 2.3 and Figure 2.5 respectively, $D(\vec{B}_4, 1) = \{1, \vec{B}_5, \vec{B}_6, 0, \vec{B}_7\}$ and $D(\vec{B}_3, 3) = \{3, \vec{B}_0, \vec{B}_1, \vec{B}_4, 1, \vec{B}_5, \vec{B}_6, 0, \vec{B}_7\}$, as shown in Figure 2.6.

Two cutpoints $u$ and $v$ of $\vec{G}$ form a *conflicting pair of cutpoints* if $u$ is an intermediate vertex in a biconnected component $\vec{B}_i$, $v$ is an intermediate vertex in a biconnected component $\vec{B}_j$, $\vec{B}_i \in D(\vec{B}_j, v)$, and $\vec{B}_j \in D(\vec{B}_i, u)$. For instance, the DAG in Figure 2.7 contains the conflicting pair of cutpoints 1 and 4. This occurs because 1 is an intermediate vertex in $\vec{B}_0$, 4 is an intermediate vertex in $\vec{B}_2$, $\vec{B}_2 \in D(\vec{B}_0, 1)$, and $\vec{B}_0 \in D(\vec{B}_2, 4)$. It should be noted that $u$ and $v$ may coincide.

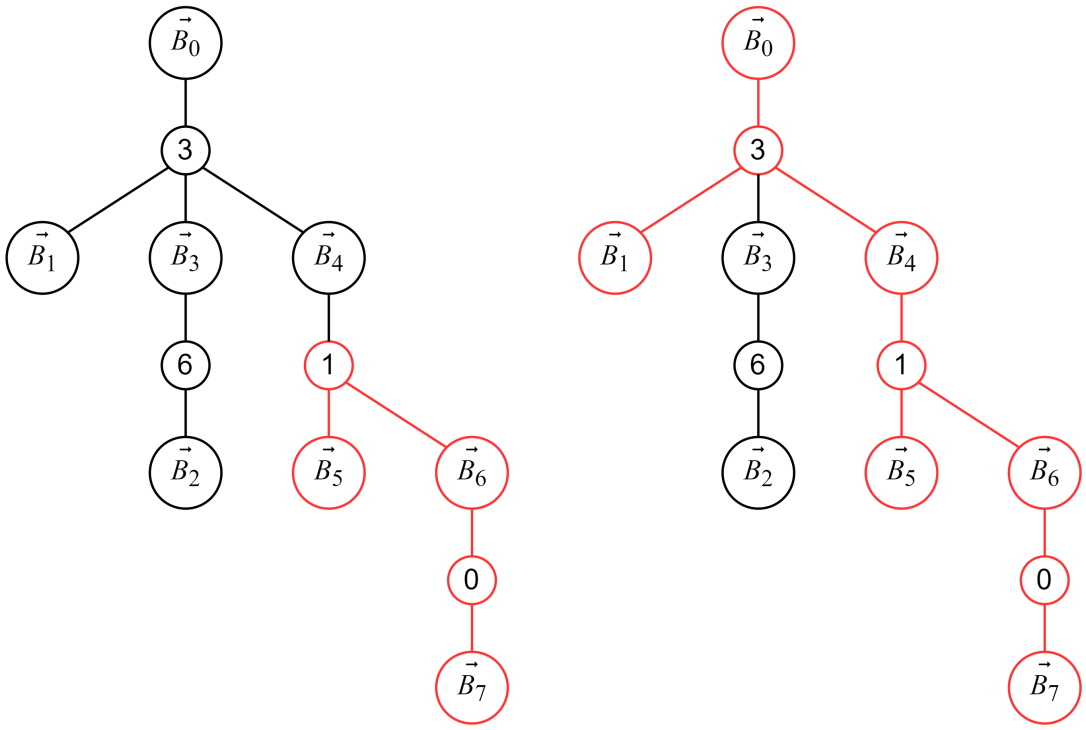

Figure 2.6: Highlighted on the left block-cutpoint tree are the nodes belonging to $D(\vec{B}_4, 1)$, while highlighted on the right block-cutpoint tree are the nodes belonging to $D(\vec{B}_3, 3)$.

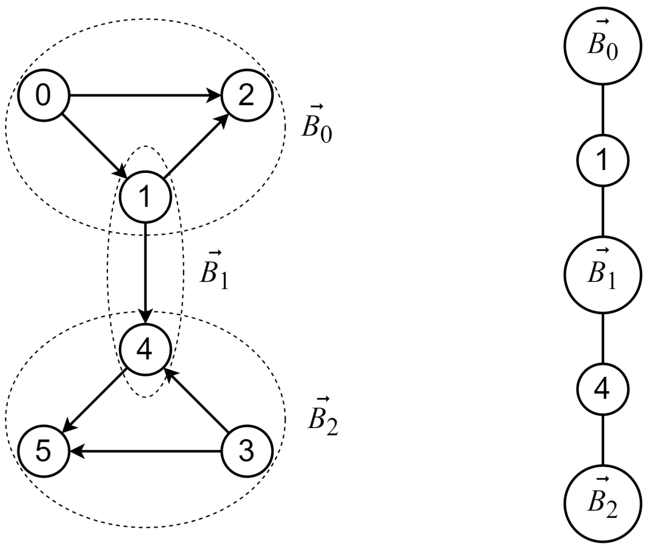

Figure 2.7: Example of a DAG with a conflicting pair of cutpoints.

The following lemma provides a partial characterization of 1-stack DAGs.

**Lemma 2** (Lemma 1.2 in [HP99])**.** *Suppose the DAG $\vec{G}$ contains a conflicting pair of cutpoints. Then $\vec{G}$ is not a 1-stack DAG.*

Let $T_r$ denote the rooted version of $T(\vec{G})$ obtained by rooting $T(\vec{G})$ at an *arbitrary biconnected component* $\vec{B}_r$. To combine the 1-stack layouts of the biconnected components of $\vec{G}$ into a 1-stack layout of $\vec{G}$, the biconnected components are first ordered through a breadth-first search of $T_r$. For example, by ordering the biconnected components of the DAG depicted in Figure 2.3 via a breadth-first search performed on the block-cutpoint tree rooted at component $\vec{B}_0$ (shown in Figure 2.5), the BFS ordering $\{\vec{B}_0, \vec{B}_1, \vec{B}_3, \vec{B}_4, \vec{B}_2, \vec{B}_5, \vec{B}_6, \vec{B}_7\}$ is obtained.

Let $\vec{B}_{[i]}$ denote the biconnected component located at the position $i$ in the previously obtained order. For each $i$, where $1 \leq i \leq m$, with $m$ being the number of biconnected components, let $\vec{G}_i$ denote the subgraph of $\vec{G}$ induced by the first $i$ biconnected components in the ordering. Specifically, $\vec{G}_1 = \vec{B}_{[1]} = \vec{B}_r$ and $\vec{G}_m = \vec{G}$.

Let $u$ be a cutpoint of $\vec{G}$, and let $\vec{B}_k$ be the parent of $u$ in $T_r$. The cutpoint $u$ is said to be *restricted* in $\vec{G}_i$ if $u$ belongs to $\vec{G}_i$, and some cutpoint $v \in D(\vec{B}_k, u)$ is an intermediate vertex in a biconnected component $\vec{B}_{[j]}$, where $j > i$ and $\vec{B}_{[j]}$ is a child of $v$. Note that it is possible that $v = u$, in which case $\vec{B}_{[j]}$ is a child of $u$. For example, in the DAG depicted in Figure 2.3, since the cutpoint 1 belongs to $D(\vec{B}_0, 3)$ and is an intermediate vertex in the biconnected component $\vec{B}_6$, cutpoint 3 is considered restricted in $\vec{G}_0$ because the biconnected component $\vec{B}_6$ is located at a position subsequent to the position of the biconnected component $\vec{B}_0$ in the BFS ordering.

The following lemma establishes an upper bound on the number of cutpoints in $\vec{B}_{[i]}$ that can be restricted in $\vec{G}_i$.

**Lemma 3** (Lemma 1.3 in [HP99])**.** *Suppose that $\vec{G}$ does not contain a conflicting pair of cutpoints. Then the number of cutpoints in $\vec{B}_{[i]}$ that are restricted in $\vec{G}_i$ is at most* 1*. Further suppose that $\vec{G}$ contains a restricted cutpoint $u$ whose parent in $T_r$ is $\vec{B}_k$. Then $u$ is either the source or the sink of $\vec{B}_k$.*

The result of [HP99] is given by the following theorem.

**Theorem 1** (Theorem 1.4 in [HP99]). *Let $\vec{G} = (V, E)$ be a DAG that does not contain a pair of conflicting cutpoints and that does not contain a biconnected component with stacknumber exceeding* 1*. Then $\vec{G}$ is a 1-stack DAG. Further, a 1-stack layout of $\vec{G}$ can be constructed in $O(|V| + |E|)$ time.*

In this theorem, the necessity is derived from Lemma 2. To demonstrate the sufficiency, an algorithm has been developed to construct a 1-stack layout of $\vec{G}$. The algorithm begins by preprocessing the biconnected components of $\vec{G}$ in the order $\vec{B}_{[m]}$, $\vec{B}_{[m-1]}$, ..., $\vec{B}_{[1]}$ to gather information about restricted vertices in each $\vec{G}_i$. Notably, $\vec{B}_{[m]}$ contains no vertices that are restricted in $\vec{G}_m$. For each $i$, where $m > i \geq 1$, the set of vertices in $\vec{B}_{[i]}$ that are restricted in $\vec{G}_i$ can be determined as follows: a vertex $u$ in $\vec{B}_{[i]}$ is a cutpoint of $\vec{G}$ and is restricted in $\vec{G}_i$ if and only if $u$ is also present in a biconnected component $\vec{B}_{[j]}$ with $j > i$, and either $u$ is an intermediate vertex in $\vec{B}_{[j]}$ or $\vec{B}_{[j]}$ contains vertices that are restricted in $\vec{G}_j$. Therefore, determining whether each cutpoint $u$ in $\vec{B}_{[i]}$ is restricted in $\vec{G}_i$ requires time proportional to the number of children $u$ has in $T_r$. Consequently, the overall preprocessing step takes time proportional to the number of edges in $T_r$. Given that the number of nodes in $T_r$ is at most $3|V| - 3$, the time complexity of this preprocessing step is $O(|V|)$. The pseudocode for this preprocessing phase is presented in Algorithm 1.

After the preprocessing step, if it is determined that some biconnected component $\vec{B}_{[i]}$ has two or more cutpoints restricted in $\vec{G}_i$, then, according to Lemma 3, $\vec{G}$ contains a conflicting pair of cutpoints, and by Lemma 2, $\vec{G}$ does not admit a 1-stack layout. Therefore, it can be assumed, without loss of generality, that after the preprocessing step, each biconnected component $\vec{B}_{[i]}$ has at most one cutpoint restricted in $\vec{G}_i$.

A vertex $u$ is considered exposed in a topological order $\sigma$ of $\vec{G}$ if there is no arc $(v, w) \in E$ such that $v <_\sigma u <_\sigma w$. An example of exposed vertices is provided in Figure 2.8.

The algorithm then processes the biconnected components of $\vec{G}$ in forward order, maintaining the following induction hypothesis.

**Induction hypothesis.** *For each $i \geq 1$, $\vec{G}_i$ has a 1-stack layout and if $u$ is restricted in $\vec{G}_i$, then $\vec{G}_i$ has a 1-stack layout in which $u$ is exposed.*

**Algorithm 1** Preprocessing Step

```
    Input: $\vec{G}, T_r, orderOfBlocks$.
    Output: $false$ if $\vec{G}$ contains a conflicting pair of cutpoints, otherwise $true$.
 1: procedure PreprocessingStep
 2:     $m \leftarrow$ number of biconnected components of $\vec{G}$
 3:     $restrictedInBlock \leftarrow$ boolean array of size $m$ initialized with all $false$ values
 4:     for $i = m - 1$ down to 0 do
 5:         $currentBlock \leftarrow orderOfBlocks[i]$
 6:         $countRestricted \leftarrow 0$
 7:         for each $cutpoint$ in $\vec{B}_{currentBlock}$ do
 8:             for each $childNode$ child of the node associated with $cutpoint$ in $T_r$ do
 9:                 $otherBlock \leftarrow$ index of the block associated with $childNode$
10:                 if $otherBlock$ is after $currentBlock$ in $orderOfBlocks$ then
11:                     if $cutpoint$ is intermediate in $\vec{B}_{otherBlock}$ or $restrictedInBlock[otherBlock]$ then
12:                         $countRestricted \leftarrow countRestricted + 1$
13:                         $restrictedInBlock[currentBlock] \leftarrow true$
14:                         $break$
15:         if $countRestricted > 1$ then
16:             return $false$
17:     return $true$
```

In order to merge the 1-stack layouts of the biconnected components of the DAG computed in Section 2.1.1 while preserving the induction hypothesis, [HP99] demonstrates that the Algorithm 2 can be used.

Since the 1-stack layout of $\vec{B}_{[i+1]}$ is constructed in linear time, as shown in Section 2.1.1, it requires $O(|\vec{B}_{[i+1]}|)$ time to extend the 1-stack layout of $\vec{G}_i$ to $\vec{G}_{i+1}$. Therefore, constructing the 1-stack layout of $\vec{G}_m$ takes a total of $O(m)$ time. Given that $m \leq |E| \leq 2|V| - 3$, the overall time complexity of the algorithm is $O(|V|)$.

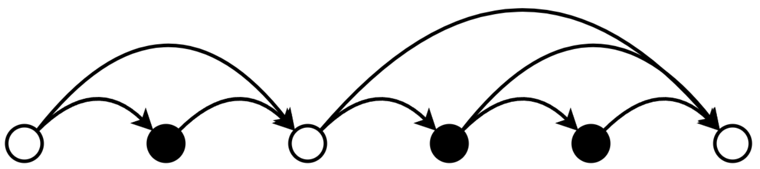

Figure 2.8: Vertices highlighted in white are exposed in the topological order shown, indicating that no arcs exist where the vertex lies between two other vertices in this order.

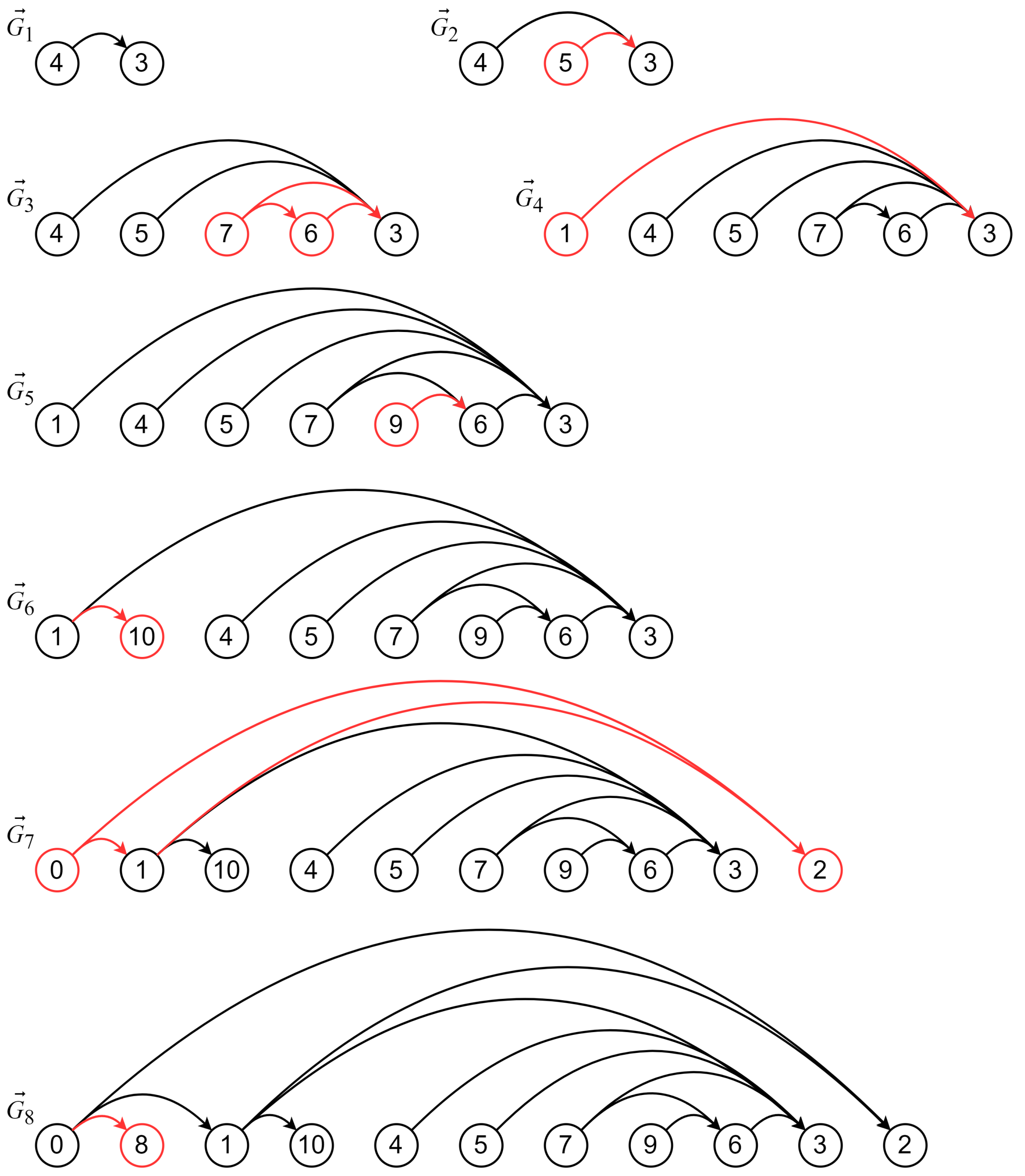

Figure 2.9: Construction of a 1-stack layout for the example DAG depicted in Figure 2.3 using the Merge1StackLayouts procedure described in Algorithm 2. For each iteration, the vertices and edges of the biconnected component being added to the current layout are highlighted in red.

**Algorithm 2** Merge 1-Stack Layouts

```
    Input: $T_r, orderOfBlocks$.
    Output: A 1-stack layout of $\vec{G}$.
 1: procedure Merge1StackLayouts
 2:     $currentBlock \leftarrow orderOfBlocks[0]$
 3:     $currentLayout \leftarrow$ the 1-stack layout of $\vec{B}_{currentBlock} : \{u_1, u_2, ..., u_t\}$
 4:     for $i = 1$ up to $m - 1$ do
 5:         $nextBlock \leftarrow orderOfBlocks[i]$
 6:         $nextLayout \leftarrow$ the 1-stack layout of $\vec{B}_{nextBlock} : \{v_1, v_2, ..., v_s\}$
 7:         $v_p \leftarrow$ the parent cutpoint of $\vec{B}_{nextBlock}$ in $T_r$ $(1 \leq p \leq s)$
 8:         $u_c \leftarrow$ the vertex $v_p$ in $currentLayout$ $(1 \leq c \leq t)$
 9:         if $v_p$ is restricted in $\vec{G}_i$ then
10:             $newLayout \leftarrow \{v_1, ..., v_{p-1}, u_1, ..., u_t, v_{p+1}, ..., v_s\}$
11:         else
12:             if $v_p$ is the source vertex of $\vec{B}_{nextBlock}$ then
13:                 $newLayout \leftarrow \{u_1, ..., u_{c-1}, v_p, v_2, ..., v_s, u_{c+1}, ..., u_t\}$
14:             else if $v_p$ is the sink vertex of $\vec{B}_{nextBlock}$ then
15:                 $newLayout \leftarrow \{u_1, ..., u_{c-1}, v_1, ..., v_p, u_{c+1}, ..., u_t\}$
16:         $currentLayout \leftarrow newLayout$
17:     return $currentLayout$
```

## 2.2 Correction of an Inaccuracy in the Algorithm

The PreprocessingStep procedure described in Algorithm 1 explores the block-cutpoint tree $T_r$, i.e., the block-cutpoint tree $T(\vec{G})$ rooted at the biconnected component $\vec{B}_r$. As stated in [HP99], this component can be any biconnected component of the input DAG $\vec{G}$. However, it has been observed that for certain DAGs, the choice of the block at which the block-cutpoint tree is rooted is critical for the correct execution of the algorithm.

For instance, the DAG shown in Figure 2.10 contains a conflicting pair of cutpoints 1 and 0. This occurs because 1 is an intermediate vertex in $\vec{B}_0$, 0 is an intermediate vertex in $\vec{B}_3$, $\vec{B}_3 \in D(\vec{B}_0, 1)$, and $\vec{B}_0 \in D(\vec{B}_3, 0)$. If, during the rooting of the block-cutpoint tree for this DAG, either the biconnected component $\vec{B}_0$ or $\vec{B}_3$ is chosen as the root, then the PreprocessingStep procedure erroneously succeds, and does not detect the conflicting pair of cutpoints in the DAG. However, if the block-cutpoint tree is rooted at $\vec{B}_1$ or $\vec{B}_2$, the procedure correctly identifies the presence of the conflicting pair of cutpoints 1 and 0 in the DAG.

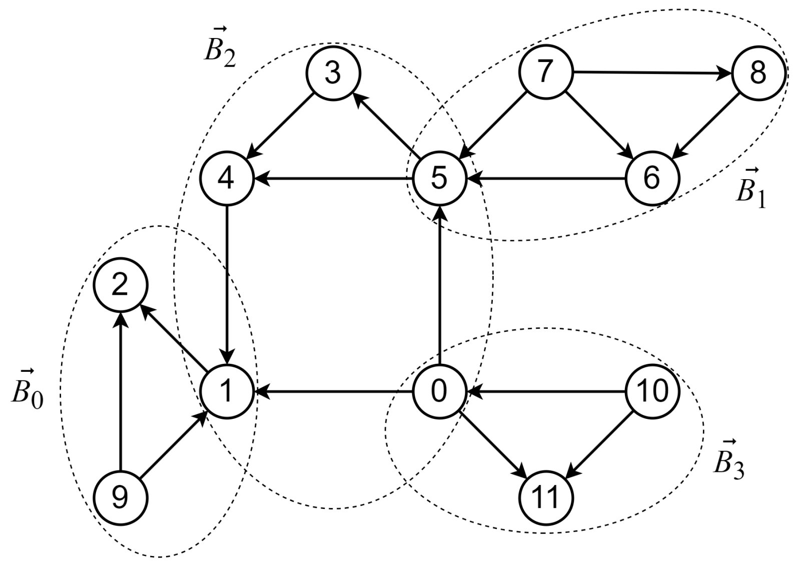

Figure 2.10: Example DAG with a conflicting pair of cutpoints.

This occurs because, in the two rootings that correctly detect the presence of the conflicting pair of cutpoints, shown in Figure 2.11, both blocks in which these cutpoints are intermediate vertices are children of the respective cutpoints. As a result, the PREPROCESSINGSTEP procedure identifies the conflicting cutpoint pair at the first common ancestor block of the two cutpoints. In contrast, in the two rootings where the algorithm erroneously succeeds despite the presence of the conflicting pair of cutpoints, shown in Figure 2.12, one of the cutpoints, either 1 or 0, is the child of the block in which it is an intermediate vertex. In this scenario, there is no common ancestor capable of detecting the conflicting pair, except for the parent block of the cutpoint itself. Thus, it becomes necessary to verify whether a restricted cutpoint that is a child of a particular block is also an intermediate vertex within that block. If this condition holds, then the input DAG contains a conflicting cutpoint pair.

To ensure that the choice of the block at which the block-cutpoint tree is rooted does not cause any variation in the algorithm's behavior, it is therefore necessary to introduce an additional condition in the procedure. This condition states that if a restricted cutpoint is found in $\vec{B}_{[i]}$ via biconnected component $\vec{B}_{[j]}$, with $j > i$, and if

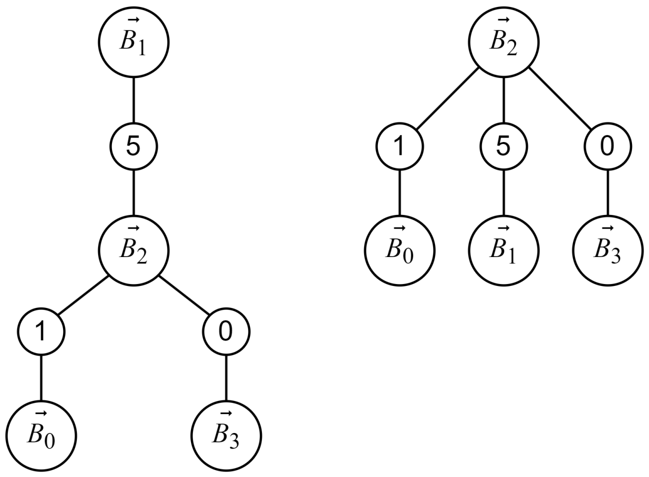

Figure 2.11: Block-cutpoint trees of the DAG in Figure 2.10 for which the preprocessing procedure returns a correct result.

this cutpoint is an intermediate vertex in $\vec{B}_i$, then a conflicting pair of cutpoints exists, and the procedure should return $false$. Algorithm 3 presents the updated version of the preprocessing procedure, in which the necessary condition to handle cases like the one just described has been added in Line 12.

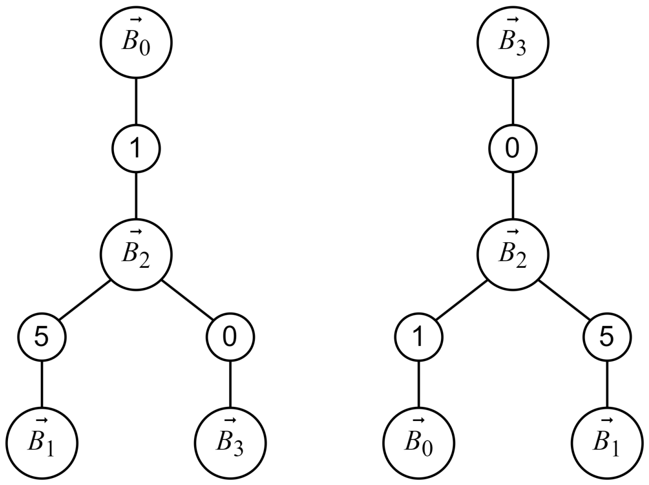

Figure 2.12: Block-cutpoint trees of the DAG in Figure 2.10 for which the preprocessing procedure returns an incorrect result.

**Algorithm 3** Preprocessing Step (Updated)

```
    Input: $\vec{G}, T_r, orderOfBlocks$.
    Output: $false$ if $\vec{G}$ contains a conflicting pair of cutpoints, otherwise $true$.
 1: procedure PreprocessingStepUpdated
 2:     $m \leftarrow$ number of biconnected components of $\vec{G}$
 3:     $restrictedInBlock \leftarrow$ boolean array of size $m$ initialized with all $false$ values
 4:     for $i = m - 1$ down to 0 do
 5:         $currentBlock \leftarrow orderOfBlocks[i]$
 6:         $countRestricted \leftarrow 0$
 7:         for each $cutpoint$ in $\vec{B}_{currentBlock}$ do
 8:             for each $childNode$ child of the node associated with $cutpoint$ in $T_r$ do
 9:                 $otherBlock \leftarrow$ index of the block associated with $childNode$
10:                 if $otherBlock$ is after $currentBlock$ in $orderOfBlocks$ then
11:                     if $cutpoint$ is intermediate in $\vec{B}_{otherBlock}$ or $restrictedInBlock[otherBlock]$ then
12:                         if $cutpoint$ is intermediate in $\vec{B}_{currentBlock}$ then
13:                             return $false$
14:                         $countRestricted \leftarrow countRestricted + 1$
15:                         $restrictedInBlock[currentBlock] \leftarrow true$
16:                         $break$
17:         if $countRestricted > 1$ then
18:             return $false$
19:     return $true$
```

## 2.3 Towards an Enumeration Algorithm

The described algorithm allows for determining whether a DAG is a 1-stack DAG. During this verification, a specific 1-stack layout of the DAG is constructed to demonstrate that it meets the 1-stack criteria. In the following chapters, it is shown how this algorithm can serve as the foundation for developing an algorithm that generates all possible 1-stack layouts of a connected DAG.

Chapter 3

# A Linear-Delay Algorithm for Enumerating 1-Stack Layouts

In this chapter, an algorithm will be presented that systematically enumerates all possible 1-stack layouts of a connected DAG, with a linear delay between the generation of each result. The algorithm is grounded in the results and methods outlined in Chapter 2, which have been enanched to facilitate the enumeration process. The distinct 1-stack layouts are produced by varying the rootings of the block-cutpoint tree associated with the DAG and by applying permutations among selected child blocks of cutpoints that meet specific criteria.

## 3.1 The Algorithm

The initial phase of the algorithm for enumerating 1-stack layouts of a connected DAG corresponds to the first step of the algorithm described in Section 2.1.1. Subsequently, the algorithm proceeds with rooting the block-cutpoint tree at an arbitrary block *rootBlock*. To indicate that the block-cutpoint tree is rooted at a specific block *rootBlock*, the notation $T_{rootBlock}$ is used.

The sections of this chapter describe the steps involved in enumerating all 1-stack layouts of a connected DAG. Section 3.1.1 presents a labeling system for the edges of the block-cutpoint tree, used to clarify the relationships between blocks and their cutpoints in the DAG. Section 3.1.2 refines the initial rooting of the block-cutpoint tree to eliminate the restrictions on cutpoints, simplifying the enumeration process. Section 3.1.3 focuses on restricted paths and discusses the conditions under which block permutations can generate new 1-stack layouts. Section 3.1.4 outlines the process for generating different BFS orderings of blocks with a linear delay between results. Finally, Section 3.1.5 integrates these steps into the full algorithm for enumerating all possible 1-stack layouts, achieving a linear delay in output generation.

### 3.1.1 Labeling of the Edges of the Block-Cutpoint Tree

The next step of the algorithm is to label the edges of the block-cutpoint tree according to the relationship between the block and the cutpoint they connect. Specifically, the labels indicate whether the cutpoint acts as a source vertex (label 0), an intermediate vertex (label 1), or a sink vertex (label 2) within the block. This labeling not only aids in streamlining the process but also facilitates the visual detection of conflicting pairs of cutpoints, which can be identified by the existence of a path between two blocks where both the first and last edges are labeled 1, as shown in Figure 3.1.

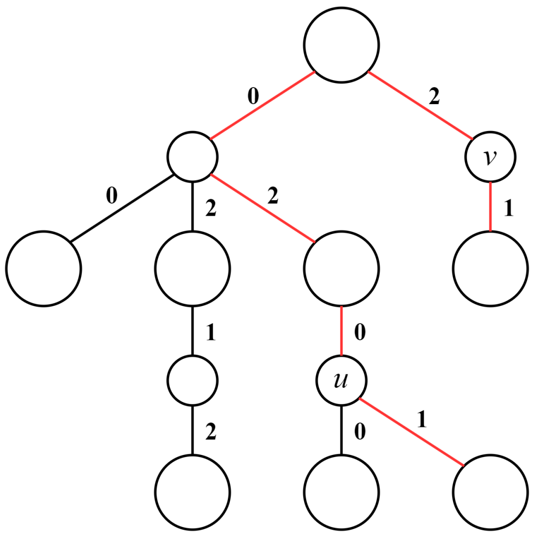

Figure 3.1: The first and last cutpoints appearing on the highlighted path, denoted as $u$ and $v$, constitute a conflicting pair of cutpoints.

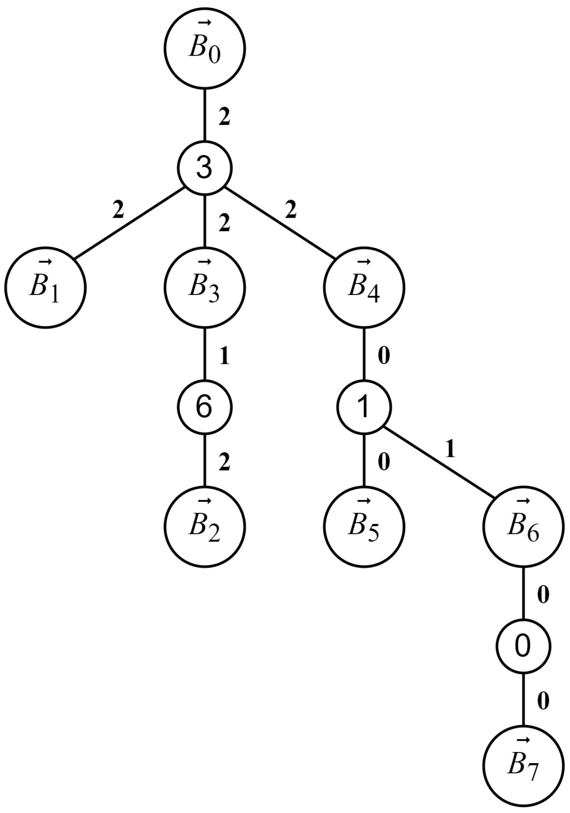

Figure 3.2: Labeled block-cutpoint tree of the DAG in Figure 2.3.

### 3.1.2 Finding a Better Initial Rooting

After constructing the block-cutpoint tree $T_{rootBlock}$, the algorithm ensures that no conflicting pairs of cutpoints exist through a revised version of the procedure described in Algorithm 3 presented in Chapter 2. This procedure, named ComputeRestrictions and described in Algorithm 4, in addition to verifying the restrictions of various cutpoints, seeks to identify a better block on which to root the block-cutpoint tree. If such a better block exists, it is referred to as $\vec{B}_{betterRoot}$ and is the first block to cause a restriction in the procedure. If the DAG does not contain a conflicting pair of cutpoints, rooting the block-cutpoint tree at $\vec{B}_{betterRoot}$ eliminates all restrictions on cutpoints, thereby simplifying subsequent calculations. For example, in the DAG shown in Figure 2.3, if the block initially chosen as the root of the block-cutpoint tree is block $\vec{B}_0$, so $rootBlock = 0$, the resulting block-cutpoint tree is $T_0$, as depicted in Figure 3.2. Since

**Algorithm 4** Compute Restrictions

**Input:** $\vec{G}, T_r, orderOfBlocks$.

**Output:** A pair of values. The first is $false$ if $\vec{G}$ contains a conflicting pair of cutpoints, otherwise $true$. The second is the better root if the procedure finds one, otherwise $-1$.

```
procedure ComputeRestrictions
    $m \leftarrow$ number of biconnected components of $\vec{G}$
    $restrictedInBlock \leftarrow$ boolean array of size $m$ initialized with all $false$ values
    $betterRoot \leftarrow -1$
    for $i = m - 1$ down to 0 do
        $currentBlock \leftarrow orderOfBlocks[i]$
        $countRestricted \leftarrow 0$
        for each $cutpoint$ in $\vec{B}_{currentBlock}$ do
            for each $childNode$ child of the node associated with $cutpoint$ in $T_r$ do
                $otherBlock \leftarrow$ index of the block associated with $childNode$
                if $otherBlock$ is after $currentBlock$ in $orderOfBlocks$ then
                    if $cutpoint$ is intermediate in $\vec{B}_{otherBlock}$ or $restrictedInBlock[otherBlock]$ then
                        if $cutpoint$ is intermediate in $\vec{B}_{currentBlock}$ then
                            return $false, -1$
                        $countRestricted \leftarrow countRestricted + 1$
                        $restrictedInBlock[currentBlock] \leftarrow true$
                        if $betterRoot = -1$ then
                            $betterRoot \leftarrow otherBlock$
                        $break$
        if $countRestricted > 1$ then
            return $false, -1$
    return $true, betterRoot$
```

block $\vec{B}_6$ is the first block, during the ComputeRestrictions procedure, to cause a restriction on a cutpoint (cutpoint 1 in this case), the *betterRoot* value returned at the end of the procedure will be 6. The algorithm then proceeds to root the block-cutpoint tree at block $\vec{B}_6$, resulting in $T_6$ as the initial block-cutpoint tree, depicted in Figure 3.3.

### 3.1.3 Restricted Paths and Permutations of Blocks

Let a *restricted path* be defined as a directed path in the block-cutpoint tree that, starting from a given cutpoint, ends at a block via an edge labeled 1. If such a path exists for a particular cutpoint, then there is a rooting of the block-cutpoint tree in

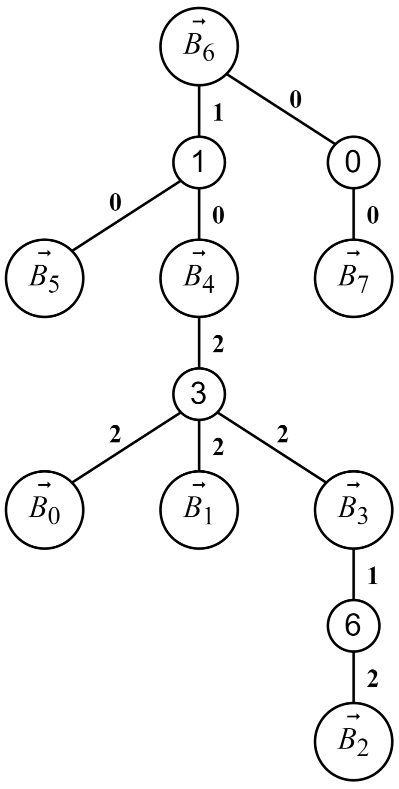

Figure 3.3: Block-cutpoint tree $T_6$ obtained after re-rooting the block-cutpoint tree at block $\vec{B}_6$, following the identification of 6 as the *betterRoot* during the ComputeRestrictions procedure. This re-rooting eliminates any cutpoint restrictions, simplifying subsequent computations.

which the cutpoint is restricted in some $\vec{G}_i$.

In order to obtain all the distinct 1-stack layouts of a given DAG, it was observed that certain blocks need to be permuted in the BFS ordering used by the Merge1StackLayouts procedure. This procedure merges the various 1-stack layouts corresponding to the biconnected components of the DAG according to a specific input block ordering. Specifically, the blocks whose permutations result in a valid block ordering and subsequently generate a valid 1-stack layout are those connected to the same cutpoint, where the edges linking them in the block-cutpoint tree are labeled 0 or 2. More precisely, it is possible to permute all blocks for which a specific cutpoint acts as a source, as well as all blocks for which the cutpoint acts as a sink. This holds except in

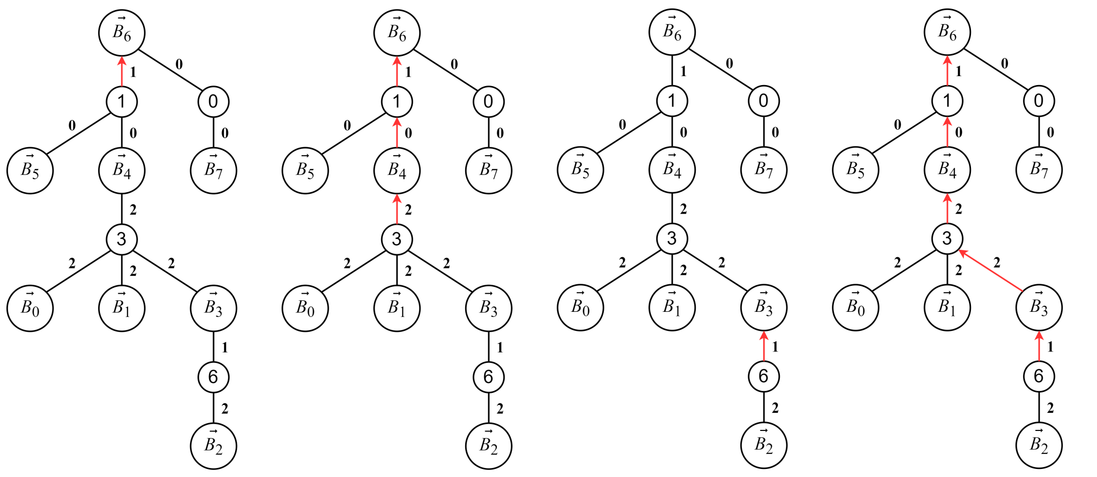

Figure 3.4: Highlighted in red are four distinct restricted paths in the block-cutpoint tree depicted in Figure 3.3. Note that all restricted paths are directed towards the root, as this block-cutpoint tree is rooted at block $\vec{B}_{betterRoot}$.

cases where there is a restricted path originating from the cutpoint and leading toward one of the blocks to be permuted. If such a restricted path exists, permuting that block would not produce a new 1-stack layout.

In the example considered, three distinct cutpoints can be identified that are adjacent to sets of blocks which, when permuted in the block ordering, result in a new 1-stack layout. In Figure 3.5, block $\vec{B}_4$ is not highlighted as permutable for cutpoint 3. This is because, as shown in Figure 3.4, there is a restricted path originating from cutpoint 3 and involving block $\vec{B}_4$. The highlighted blocks, adjacent to cutpoints 0, 1, and 3 in Figure 3.5, generate new 1-stack layouts when permuted. If all permutable blocks are child blocks of the common cutpoint in the rooted block-cutpoint tree $T_{betterRoot}$, permuting these child blocks in the BFS ordering of biconnected components can be done by adjusting the order in which the child blocks are added to the queue during a breadth-first search on $T_{betterRoot}$ (as is the case for cutpoints 1 and 3 in the example block-cutpoint tree). However, if the parent block in $T_{betterRoot}$ of the given cutpoint is also part of the set of permutable blocks (cutpoint 0 in the example), to obtain an ordering in which a child block appear before the parent block, a breadth-first search can be performed on the block-cutpoint tree rooted at one of the permutable child blocks.

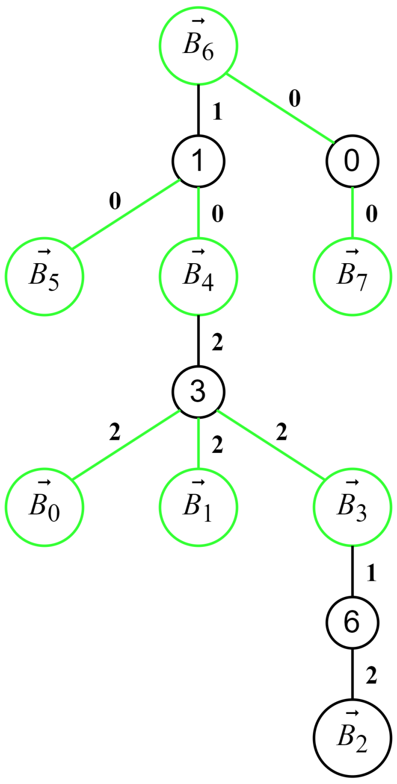

Figure 3.5: Highlighted in green are the permutable blocks identified in the example block-cutpoint tree $T_{betterRoot}$.

In the example, a breadth-first search on $T_7$ produces a BFS ordering where block $\vec{B}_7$ appears before block $\vec{B}_6$.

Based on the previous considerations, the algorithm proceeds by searching for blocks suitable for rooting using the FindOtherRoots procedure described in Algorithm 5. This procedure traverses the block-cutpoint tree $T_{betterRoot}$, looking for cutpoints that act as a source (or sink) for both the parent block and one or more child blocks. When such a cutpoint is found, its child blocks are added to the list of root blocks to be considered. Furthermore, to avoid selecting permutable blocks belonging to restricted paths originating from the child cutpoint, the procedure skips type-1 edges that lead from a block to a child cutpoint in $T_{betterRoot}$. Through this procedure, the list of root blocks that can generate new results is obtained in linear time.

**Algorithm 5** Find Other Roots

```
Input: $rootNode, rootBlock, parentType$.
Output: A list of alternative root blocks for the BCT.
procedure FindOtherRoots
    if $rootNode$ is a leaf of $T_{rootBlock}$ then
        return an empty list
    if $rootNode$ is a cutpoint then
        $otherRoots \leftarrow$ an empty list
        for $childType = 0$ up to 2 do
            for each $child$ of rootNode of type $childType$ in $T_{rootBlock}$ do
                extend $otherRoots$ with FindOtherRoots($child$, $rootBlock$, $childType$)
                if $childType = parentType$ then
                    add the index of $child$ to $otherRoots$
        return $otherRoots$
    else
        $otherRoots \leftarrow$ an empty list
        for each $child$ of $rootNode$ of type 0 in $T_{rootBlock}$ do
            extend $otherRoots$ with FindOtherRoots($child$, $rootBlock$, 0)
        for each $child$ of $rootNode$ of type 2 in $T_{rootBlock}$ do
            extend $otherRoots$ with FindOtherRoots($child$, $rootBlock$, 2)
        return $otherRoots$
```

### 3.1.4 The Enumeration Process

To compute all possible 1-stack layouts of the DAG $\vec{G}$ with a linear delay between each layout, it is necessary to obtain a new BFS ordering of blocks that produces, via the Merge1StackLayouts procedure, a new 1-stack layout of $\vec{G}$ in linear time. To achieve this, the algorithm employs instances from a struct called *BCTPermutationEnumerator*. This struct enables the generation of different block orderings that can produce new 1-stack layouts through breadth first search on a block-cutpoint tree rooted at a specific block. The struct defines three variables and three procedures, as described in Algorithm 6. The most important variable in the struct is *permutations*, a map of maps that stores the state of permutations for the children of a certain type of a specific cutpoint, allowing the current state to be used to derive the next one.

To initialize an instance of the *BCTPermutationEnumerator* struct, the InitializeEnumerator procedure must be called, which takes as input only a block-cutpoint

**Algorithm 6** BCTPermutationEnumerator Struct Definition

```
struct BCTPermutationEnumerator
    $T_{rootBlock}$ : The rooted block cutpoint tree on which the enumeration must be performed.
    $permutations$ : A nested map containing lists of child indices to be permuted. The first key represents the parent cutpoint, and the second key specifies the type of the children.
    $hasNextPermutation$ : A boolean variable indicating whether a subsequent permutation exists.

    procedure InitializeEnumerator($T_{rootBlock}$)
    procedure HasNextPermutation
    procedure GetNextPermutation
end struct
```

tree rooted at a specific block. The initialization procedure, described in Algorithm 7, performs a breadth-first search on the block-cutpoint tree to identify cutpoints that reach more than one child block through a type 0 or type 2 edge, meaning the children for which the cutpoint acts as a source (type 0 children) or as a sink (type 2 children). When a set of more than one child of the same type (0 or 2) is found at a cutpoint, a state for that set is initialized in the *permutations* variable with the cutpoint and the child type as keys. This initial state represents the first permutation of the indices of the list of the $k$ children of the same type and is set to $\{0, 1, ..., k-1\}$, which corresponds to the current order in which they are stored in the list. Thus, the initialization procedure sets up the variables of the *BCTPermutationEnumerator* struct and saves the initial states of the permutable child sets of the various cutpoints in the rooted block-cutpoint tree.

To generate block orderings with a linear delay between one ordering and the next, the *BCTPermutationEnumerator* performs a breadth-first search on the associated block-cutpoint tree and inserts the permutable child blocks of each cutpoint into the BFS ordering, following the index order described in the *permutations* map. Afterward, if a next permutation exists for the index list associated with the current set of permutable children, this is generated and replaces the current index list in *permutations*; otherwise, the current index list is reinitialized. This index list modification is performed only once, specifically on the first list that allows for a next permutation. By

**Algorithm 7** Initialize Enumerator

```
    Input: $T_{rootBlock}$.
    Output: An initialized $BCTPermutationEnumerator$.
 1: procedure InitializeEnumerator($T_{rootBlock}$)
 2:     $T_{rootBlock} \leftarrow T_{rootBlock}$
 3:     $permutations \leftarrow$ an empty map
 4:     $hasNextPermutation \leftarrow true$
 5:     $queue \leftarrow$ an empty queue
 6:     $queue.push$(the root node of $T_{rootBlock}$)
 7:     while $queue$ is not empty do
 8:         $u \leftarrow queue.pop()$
 9:         if $u$ is a cutpoint then
10:             $sourceChildren \leftarrow$ list of child of type 0 of $u$ in $T_{rootBlock}$
11:             if $sourceChildren.length > 1$ then
12:                 $firstPermutation \leftarrow \{0, 1, ..., sourceChildren.length - 1\}$
13:                 $permutations$[index of $u$][0] $\leftarrow firstPermutation$
14:             $sinkChildren \leftarrow$ list of child of type 2 of $u$ in $T_{rootBlock}$
15:             if $sinkChildren.length > 1$ then
16:                 $firstPermutation \leftarrow \{0, 1, ..., sinkChildren.length - 1\}$
17:                 $permutations$[index of $u$][2] $\leftarrow firstPermutation$
18:         for each $child$ of $u$ in $T_{rootBlock}$ do
19:             $queue.push(child)$
20:     return the initialized $BCTPermutationEnumerator$
```

"next permutation", this refers to the permutation that follows in lexicographical order. This operation can be executed in linear time relative to the length of the list to be permuted, using the algorithm described in [Knu11] and represented by the NextPermutation procedure in Algorithm 8, which returns *true* if the next permutation has been computed and *false* otherwise. Since the index list is initially ordered, obtaining the next permutation gradually yields all possible permutations of those indices, and consequently all permutations of the permutable child lists. The procedure described in Algorithm 8 not only provides the next block ordering, but also detects when all possible permutations have been exhausted, setting the *hasNextPermutation* variable to *false* if none of the index lists admits a next permutation.

**Algorithm 8** Get Next Permutation

```
    Input: nothing.
    Output: The next permutation of blocks.
 1: procedure GetNextPermutation
 2:     $toPermute \leftarrow true$
 3:     $order \leftarrow$ an empty list
 4:     $queue \leftarrow$ an empty queue
 5:     $queue.push$(the root node of $T_{rootBlock}$)
 6:     while $queue$ is not empty do
 7:         $u \leftarrow queue.pop()$
 8:         if $u$ is a cutpoint then
 9:             for each $childType$ in $\{0, 2\}$ do
10:                 $children \leftarrow$ list of child of type $childType$ of $u$ in $T_{rootBlock}$
11:                 if $children.length > 1$ then
12:                     $permutation \leftarrow permutations$[index of $u$][$childType$]
13:                     for $i = 0$ up to $permutation.length - 1$ do
14:                         $indexOfChild \leftarrow permutation[i]$
15:                         $queue.push(children[indexOfChild])$
16:                     if $toPermute$ then
17:                         if NextPermutation($permutation, u, childType$) then
18:                             $toPermute \leftarrow false$
19:                         else
20:                             $firstPerm \leftarrow \{0, 1, ..., children.length - 1\}$
21:                             $permutations$[index of $u$][$childType$] $\leftarrow firstPerm$
22:                 else
23:                     if $children.length = 1$ then
24:                         $queue.push(children[0])$
25:         else
26:             append the index of $u$ to $order$
27:             for each $child$ of $u$ in $T_{rootBlock}$ do
28:                 $queue.push(child)$
29:     if toPermute then
30:         $hasNextPermutation \leftarrow false$
31:     return $order$
```

### 3.1.5 The Final Algorithm

Now that all the necessary procedures for enumerating all possible 1-stack layouts of a given DAG $\vec{G}$ have been described, the entire enumeration process is outlined in Algorithm 9. The enumeration procedure first verifies that the DAG is outerplanar and that each biconnected component contains a Hamiltonian path along the outer face. Subsequently, the procedure roots the associated block-cutpoint tree at component $\vec{B}_0$ and uses the ComputeRestrictions procedure to check for conflicting pairs of cutpoints within the DAG, as well as to search for a better rooting for the block-cutpoint tree, that allows for a tree without restricted cutpoints. If such a rooting exists, it is chosen as the initial rooting of the block-cutpoint tree.

At this point, the FindOtherRoots procedure is employed to identify additional potential rootings of the block-cutpoint tree that can yield new 1-stack layouts, where the root component appears before the others in the BFS ordering of blocks. An instance of struct *BCTPermutationEnumerator* is then initialized for the block-cutpoint tree with the initial rooting, and the possible permutations of the blocks are generated through the GetNextPermutation procedure. The resulting 1-stack layout is generated using the Merge1StackLayouts procedure, applied to both the initial rooting and the obtained BFS ordering of blocks. Finally, if there are other valid rootings of the block-cutpoint tree, all 1-stack layouts for these rootings are similarly generated. This approach enables the complete enumeration of all 1-stack layouts for a given DAG $\vec{G}$ with a delay of $O(n)$, where $n$ is the number of vertices in $\vec{G}$.

## 3.2 Counting the Number of 1-Stack Layouts

After rooting the block-cutpoint tree at the better root and determining the set of all possible rootings $R$, containing the initial rooting and the others rootings derived through the FindOtherRoots procedure, it becomes possible to compute the number of 1-stack layouts that the input DAG can generate. The number of 1-stack layouts obtainable from the algorithm, starting from a specific rooting of the block-cutpoint tree, is determined by multiplying the numbers of permutations among all sets of blocks that are children of a cutpoint in the given rooting and are permutable with each other.

**Algorithm 9** Enumerate 1-Stack Layouts

```
Input: A connected DAG $\vec{G}$.
Output: All possible 1-stack layouts of $\vec{G}$
procedure ENUMERATE1STACKLAYOUTS
    if $\vec{G}$ is not outerplanar then
        $reportfailure$
    for each biconnected component $\vec{B}_i$ in $\vec{G}$ do
        if $\vec{B}_i$ does not have an Hamiltonian path on the outer face then
            $reportfailure$
    $rootBlock \leftarrow 0$
    $orderOfBlocks \leftarrow$ list of block indexes in BFS order of $T_{rootBlock}$
    $noConflictingPair, betterRoot \leftarrow$ COMPUTERESTRICTIONS($\vec{G}$, $orderOfBlocks$)
    if not $noConflictingPair$ then
        $reportfailure$
    if $betterRoot \neq$ -1 then
        $rootBlock \leftarrow betterRoot$
    $otherRoots \leftarrow$ FINDOTHERROOTS(the root node of $T_{rootBlock}, rootBlock, -1$)
    $enumerator \leftarrow$ INITIALIZEENUMERATOR($T_{rootBlock}$)
    while $enumerator$.HASNEXTPERMUTATION() do
        $orderOfBlocks \leftarrow enumerator$.GETNEXTPERMUTATION()
        $result \leftarrow$ MERGE1STACKLAYOUTS($T_{rootBlock}, orderOfBlocks$)
        $print\ result$
    for each $otherRoot$ in $otherRoots$ do
        $enumerator \leftarrow$ INITIALIZEENUMERATOR($T_{otherRoot}$)
        while $enumerator$.HASNEXTPERMUTATION() do
            $orderOfBlocks \leftarrow enumerator$.GETNEXTPERMUTATION()
            $result \leftarrow$ MERGE1STACKLAYOUTS($T_{otherRoot}, orderOfBlocks$)
            $print\ result$
```

Let $P(T_i)$ be the set containing all the sets of blocks that are children of a cutpoint and permutable with each other in $T_i$; if no such sets exist in $T_i$, then $P(T_i)$ contains only a set $p_0$ of size 0. Since the number of possible permutations in a set of $n$ elements is $n!$, the total number of 1-stack layouts that the input DAG can generate is given by the following formula:

$$\text{number of 1-stack layouts} = \sum_{\vec{B}_i \in R} \prod_{p \in P(T_i)} |p|!$$

In the example DAG, the number of block-cutpoint tree there are two rootings to be considered, as the procedure FINDOTHERROOTS identifies an additional rooting in block $\vec{B}_7$, besides the initial rooting in $\vec{B}_6$. Figure 3.6 highlights the sets of child blocks of a cutpoint that are permutable with each other in both rootings. It can be observed that the size of these sets remains the same for every rooting considered. Therefore, since cutpoint 1 has two permutable child blocks and cutpoint 3 has three permutable child blocks, the total number of 1-stack layouts that can be generated from the example DAG is $(2! \cdot 3!) + (2! \cdot 3!) = 24$.

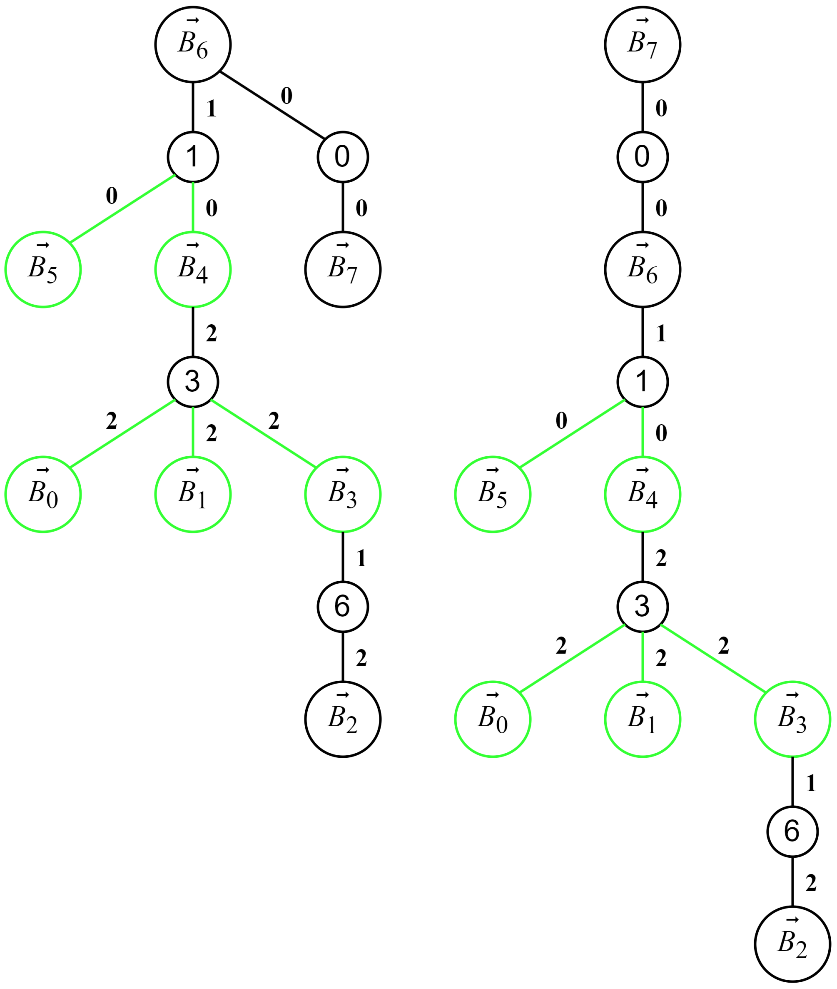

Figure 3.6: Highlighted in green are the child blocks of a cutpoint that are permutable in the two rootings obtained for the example DAG.

## 3.3 Correctness of the Algorithm

This section is devoted to the proof of the following main result.

**Theorem 2.** *The Algorithm* Enumerate1StackLayouts *enumerates all 1-stack layouts of an $n$-vertex input DAG without repetitions with worst-case $O(n)$ delay.*

To prove Theorem 2, Lemmas 4 to 6 are used. In particular, Lemma 4 establishes that all 1-stack layouts generated by the Enumerate1StackLayouts algorithm correspond to valid 1-stack layouts of the input DAG. Lemma 5 shows that all 1-stack layout produced by the Enumerate1StackLayouts algorithm are distinct. Also, Lemma 6 establishes that the Enumerate1StackLayouts algorithm lists all valid 1-stack layouts of the input DAG.

Given an input DAG $\vec{G} = (V, \vec{E})$, the Enumerate1StackLayouts algorithm requires $O(|V|)$ time to generate a BFS ordering of the blocks starting from a specified root of the block-cutpoint tree. Once this ordering is obtained, it serves as input to the Merge1StackLayouts procedure, which also operates in $O(|V|)$ time. Additionally, the initialization phase of a *BCTPermutationEnumerator* takes $O(|V|)$ time. Based on these factors, it can be stated that the delay between generating a 1-stack layout of $\vec{G}$ and the next, using the Enumerate1StackLayouts algorithm, is $O(|V|)$.

The proof starts with Lemma 4, which asserts that if the algorithm successfully generates a 1-stack layout $\alpha$ for a given DAG $\vec{G}$, then $\alpha$ is indeed a valid 1-stack layout of $\vec{G}$. This lemma emphasizes the reliability of the algorithm in producing layouts that adhere to the structural constraints imposed by the graph.

**Lemma 4.** *Let $\vec{G}$ be a directed acyclic graph (DAG), and let $\alpha$ be an ordering of the vertices of $\vec{G}$ generated by the algorithm* Enumerate1StackLayouts*. Then, $\alpha$ is a valid 1-stack layout of $\vec{G}$.*

*Proof.* The algorithm Enumerate1StackLayouts generates layouts by permuting child blocks of cutpoints in some rooting of the block-cutpoint tree of $\vec{G}$, using procedures that are based on well-established methods described in [HP99]. Specifically, the steps for verifying and preprocessing the input DAG, as well as the procedure for merging 1-stack layouts of its biconnected components, follow the same approach as in

[HP99], where it was proven that these procedures always yield a valid 1-stack layout of $\vec{G}$. The only modifications made by the ENUMERATE1STACKLAYOUTS algorithm in generating each 1-stack layout pertain to the rooting of the block-cutpoint tree and the permutations of blocks in determining the block order. Since the algorithm described in [HP99] allows any block of the input DAG $\vec{G}$ to serve as the root of the block-cutpoint tree, and since the block permutations involve only the child blocks of cutpoints in the considered rooting of the block-cutpoint tree, which ensures that these permutations still form valid BFS orderings, if the ENUMERATE1STACKLAYOUTS algorithm produces a 1-stack layout $\alpha$, then $\alpha$ is a valid 1-stack layout of $\vec{G}$. □

Lemma 5 ensures that the algorithm avoids duplication, proving that it never generates the same layout more than once.

**Lemma 5.** *The algorithm ENUMERATE1STACKLAYOUTS does not generate the same 1-stack layout more than once.*

*Proof.* The proof begins with the following claim: The ENUMERATE1STACKLAYOUTS algorithm, through the procedure COMPUTERESTRICTIONS, finds a root block $\vec{B}_{root}$ such that the block-cutpoint tree $T_{root}$ has no restrictions on the cutpoints, all child blocks of cutpoints are either type 0 or type 2.

Proof of the claim: Assuming for contradiction that there exists a type 1 child block $\vec{B}_1$ of a cutpoint $C_1$ in $T_{root}$, two cases arise:

1. COMPUTERESTRICTIONS found a better root: In this case, the initial rooted block-cutpoint tree $T_r$ has been replaced with $T_{root}$. Let $C_0$ be the child cutpoint in $T_{root}$ of the block $\vec{B}_{root}$ that is intermediate in this block. Two cases are then possible:

   - If $\vec{B}_1 \in D(\vec{B}_{root}, C_0)$, then $C_0$ and $C_1$ would form a conflicting pair of cutpoints, as $\vec{B}_{root}$ would also be in $D(\vec{B}_1, C_1)$. Thus, the COMPUTERESTRICTIONS procedure would not have found a better root because the input DAG does not admit a 1-stack layout.
   - If $\vec{B}_1 \notin D(\vec{B}_{root}, C_0)$, then $C_0$ and $C_1$ would not form a conflicting pair of cutpoints. However, during the COMPUTERESTRICTIONS procedure, block

$\vec{B}_1$ would have been selected as the better root since it is the first block to cause a restriction for any initial rooted block-cutpoint tree $T_r$.

Let $C_t$ be the child cutpoint in $T_{root}$ of the block $\vec{B}_{root}$ such that $\vec{B}_1 \in D(\vec{B}_{root}, C_t)$. If $B_r$ were any block in $D(\vec{B}_{root}, C_i)$ with $C_i \neq C_t$, then block $\vec{B}_1$ would be at a greater depth than $\vec{B}_{root}$ in $T_r$ and therefore would be processed earlier in the ComputeRestrictions procedure, which processes blocks in reverse order of a BFS traversal of $T_r$. Thus, $\vec{B}_1$ would be the better root.

If instead $\vec{B}_r \in D(\vec{B}_{root}, C_t)$, the only block causing a cutpoint restriction is $\vec{B}_1$, as in this rooting, $C_0$ would not be restricted since it no longer has a child block in which it is intermediate, since $\vec{B}_{root}$ is the parent of $C_0$ in this rooting. Hence, $\vec{B}_1$ would be the better root.

2. ComputeRestrictions did not find a better root: In this case, $T_r = T_{root}$, meaning the ComputeRestrictions procedure did not find any restricted cutpoint in $T_r$ that would lead to the selection of a better root. Therefore, $\vec{B}_1$ cannot be a type 1 child of $C_1$, as otherwise, $C_1$ would be restricted since it is intermediate in the child block $\vec{B}_1$.

Since all alternative rootings are selected by the procedure FindOtherRoots by visiting $T_{root}$ while ensuring that no type 1 edge from a block to a cutpoint is traversed, every other alternative block-cutpoint tree rooting $T_{otherRoot}$ found by this procedure has the same property as $T_{root}$ of containing no restricted cutpoint.

For each rooting $T_i$ to be considered, that is $T_{root}$ and the rootings found through the FindOtherRoots procedure, there are no restricted cutpoints. Therefore, in the Merge1StackLayouts procedure, the only possible case is that when the layout of a new block $\vec{B}_{new}$ is added to the current layout, the cutpoint $C_p$, parent of $\vec{B}_{new}$ in $T_i$, is either the source or the sink of $\vec{B}_{new}$. If $C_p$ is the source of $\vec{B}_{new}$, the procedure indicates that the vertices of $\vec{B}_{new}$ should be inserted in the order of the 1-stack layout of $\vec{B}_{new}$ (excluding the source) immediately to the right of $C_p$ in the current layout. Conversely, if $C_p$ is the sink of $\vec{B}_{new}$, the procedure indicates that the vertices of $\vec{B}_{new}$ should be inserted in the order of the 1-stack layout of $\vec{B}_{new}$ (excluding the sink) immediately to the left of $C_p$ in the current layout. Thus, if cutpoint $C_p$ has both a type 0 child block $\vec{B}_0$ and a type 2 child block $\vec{B}_2$ in $T_i$, the order in which these are added to the current layout does not affect the result, as in either case, the vertices of $\vec{B}_0$ would be to the right of $C_p$ and the vertices of $\vec{B}_2$ would be to the left of $C_p$. Swapping the position of $\vec{B}_0$ with $\vec{B}_2$ in the BFS ordering input for the MERGE1STACKLAYOUTS procedure would therefore result in the generation of the same 1-stack layout, leading to a collision.

However, if cutpoint $C_p$ has multiple type 0 child blocks in $T_i$, the first of these children added to the current layout is placed immediately to the right of $C_p$, but subsequently, the first added block would always be farther from $C_p$ as the vertices of sibling blocks of the same type are added immediately to the right of $C_p$. Conversely, if $C_p$ has more than one type 2 child block in $T_i$, the first added block would be to the left of $C_p$ in the layout, moving farther as additional sibling blocks of type 2 are added.

Therefore, for two different BFS orderings of $T(\vec{G})$ to produce two different 1-stack layouts, the relative orderings among the permutable blocks, neighbors of the same cutpoint in $T(\vec{G})$ must differ; otherwise, a collision occurs.

Since the algorithm ENUMERATE1STACKLAYOUTS constructs different BFS orderings such that at least one set of permutable blocks in $T(\vec{G})$ has a different relative ordering compared to the previously constructed BFS orderings, the algorithm ENUMERATE1STACKLAYOUTS does not generate the same 1-stack layout more than once. □

Lemma 6 complements Lemma 4 by stating that if a valid 1-stack layout exists for $\vec{G}$, the algorithm will generate it. This lemma highlights the completeness of the algorithm, ensuring that all potential layouts are enumerated. Together, these lemmas provide a theoretical foundation for the algorithm, confirming its efficacy in both generating valid layouts and exploring the entirety of the solution space.

**Lemma 6.** *Let $\vec{G}$ be a directed acyclic graph (DAG) that admits a 1-stack layout. Then, the algorithm* ENUMERATE1STACKLAYOUTS *generates this layout.*

*Proof.* Let $\alpha$ be a valid 1-stack layout of the input DAG $\vec{G}$ and let $v_0$ be the first vertex in $\alpha$. Then, there are two possible cases for the selection of the root block $\vec{B}_0$ of the block-cutpoint tree:

1. $v_0$ is a cutpoint of $\vec{G}$: In this case, $v_0$ is neither an intermediate nor a sink in any block of $\vec{G}$, because if that were the case, it could not be the first vertex in $\alpha$. Therefore, $v_0$ appears as the source in at least two blocks, since if it were the source in only one block, it would not be a cutpoint. Let $\vec{B}_0$ be the block in which $v_0$ is the source, and whose sink is the farthest from $v_0$ in $\alpha$ compared to the sinks of the other blocks in which $v_0$ is the source.

2. $v_0$ is not a cutpoint of $\vec{G}$: In this case, let $\vec{B}_0$ be the block in which $v_0$ is the source.

Since the vertex $v_0$ is the source of the block $\vec{B}_0$ and $v_0$ is the first vertex in $\alpha$, there can be no restricted cutpoints in $T_0$, because if there were a cutpoint $C_1$ in $T_0$ such that $C_1$ is intermediate in its child block $\vec{B}_1$, then the source vertex of $\vec{B}_1$ would necessarily appear before $v_0$ in $\alpha$, which is impossible since $v_0$ is the first vertex in $\alpha$.

Let $\beta$ be the ordering of the blocks obtained by performing a BFS on $T_0$ and following these rules:

- When traversing a cutpoint $C_i$ that has more than one type-0 child block, these child blocks are explored in order of decreasing distance of their sink from $C_i$ in $\alpha$. That is, the block whose sink is farther from $C_i$ in $\alpha$ is explored first, compared to the sinks of its sibling blocks.

- When traversing a cutpoint $C_i$ that has more than one type-2 child block, these child blocks are explored in order of decreasing distance of their source from $C_i$ in $\alpha$. That is, the block whose source is farther from $C_i$ in $\alpha$ is explored first, compared to the sources of its sibling blocks.

By building a 1-stack layout of $\vec{G}$ through the procedure Merge1StackLayouts with $T_0$ and $\beta$ as input, the procedure adds the 1-stack layouts of the blocks in the order described by $\beta$. First, the 1-stack layout of $\vec{B}_0$ is inserted, and since $\beta$ is a BFS ordering of $T_0$, the 1-stack layouts of the blocks reachable through the child cutpoints of $\vec{B}_0$ in $T_0$, are added next. For each child cutpoint $C_i$ of $\vec{B}_0$ in $T_0$, if $C_i$ has more than one child block where it is the source, then, given the way $\beta$ is constructed, the 1-stack layouts of these children will be inserted in order of decreasing distance of their sink from $C_i$

in $\alpha$. The same happens symmetrically if $C_i$ has more than one child block where it is the sink. This process continues as all the blocks in $\beta$ are added to the current layout.

When the procedure MERGE1STACKLAYOUTS adds the 1-stack layout of a new block $\vec{B}_{new}$ to the current layout, it is inserted immediately to the left or right of its parent cutpoint $C_p$ in $T_0$. This means that the relative order of the blocks in $\beta$, that are children of a cutpoint $C_i$ in $T_0$ and of the same type, is reflected in their distance from $C_i$ in the 1-stack layout generated by the procedure MERGE1STACKLAYOUTS.

Thus, the ordering $\beta$, obtained by following the rules of ordering according to the greater distance of child blocks of the same cutpoint and type, when used as input along with the block-cutpoint tree $T_0$ in the procedure MERGE1STACKLAYOUTS, generates the 1-stack layout $\alpha$.

Since the block ordering $\beta$ is a BFS ordering of $T(\vec{G})$ in which sets of permutable blocks have a specific relative ordering, and since the algorithm ENUMERATE1STACK-LAYOUTS explores all possible relative orderings of these blocks in the various BFS orderings of $T(\vec{G})$ that are generated, a BFS ordering $\beta'$ of the blocks that has the same relative ordering of permutable blocks as $\beta$ will be generated by the algorithm, leading to the construction of the 1-stack layout $\alpha$. □

# Chapter 4

# A Web System for Enumerating 1-Stack Layouts

This chapter describes the development of a web system designed to enumerate all possible 1-stack layouts for a DAG, as detailed in Chapter 3. The system allows users to interact with the enumeration algorithm through a web interface, providing a practical tool for exploring different layout possibilities. The implementation process, encountered challenges, and the custom library created to interact with the algorithm are discussed in detail.

## 4.1 Implementation of the Algorithm

The core of the system relies on an efficient C++ implementation of the enumeration algorithm, developed using the Open Graph Drawing Framework (OGDF) library [CGJ+14]. The choice of OGDF is motivated by its comprehensive support for graph manipulation and layout algorithms, making it ideal for handling the graph-related operations required by the enumeration algorithm.

The implementation of the various phases of the algorithm is encapsulated within a class named *OneStackLayoutsEnumerator*. This class requires a graph as input during its initialization. To manage the graph's structure, the *Graph* class from the OGDF library has been employed. The *OneStackLayoutsEnumerator* object is thus initialized through an instance of the *Graph* class from OGDF, after which it becomes possible to request both the total number of 1-stack layouts that can be generated by the input DAG and the next 1-stack layout. The initialization process encompasses the setup of the algorithm's stages before the enumeration begins. First, the input graph is verified for outerplanarity using auxiliary functions provided by OGDF. Subsequently, the biconnected components of the graph are computed, and, once it is confirmed that each biconnected component contains a Hamiltonian path on the outer face, the 1-stack layouts of each block are calculated through the topological sorting functionality provided by OGDF.

Following this, the block-cutpoint tree is constructed, complete with its edge labels, and the presence of any conflicting pairs of cutpoints is checked via an implementation of the ComputeRestrictions procedure described in Algorithm 4, which also provides a better initial rooting for the block-cutpoint tree, if one exists. Subsequently, the necessary additional rootings for enumerating all possible 1-stack layouts of the input DAG are found through an implementation of the FindOtherRoots procedure described in Algorithm 5, and an instance of an implementation of the *BCTPermutationEnumerator* class is initialized on the block-cutpoint tree with the initial rooting. The current enumerator, along with the other computed rootings, are stored in private variables of the *OneStackLayoutsEnumerator* class, as they are required for the generation of all possible 1-stack layouts of the graph.

When the next 1-stack layout is requested from the *OneStackLayoutsEnumerator* object, the enumerator computes the next BFS ordering of blocks through an implementation of the GetNextPermutation procedure described in Algorithm 8. The NextPermutation procedure, invoked within the GetNextPermutation procedure, is implemented using the *std::next_permutation* function from C++, which guarantees the generation of the lexicographically next permutation in $O(n)$ time in the worst-case, where $n$ is the number of elements to permute. If a valid BFS ordering exists, a new 1-stack layout is generated based on the obtained ordering through an implementation of the Merge1StackLayouts procedure described in Algorithm 2. However, if no further valid BFS ordering can be produced, the algorithm checks if there is an additional BCT rooting to consider. If such a rooting exists, a new enumerator is initialized for this rooting, and the 1-stack layout is generated as described. The total number of possible 1-stack layouts is computed during initialization by applying the formula detailed in Section 3.2.

## 4.2 WebAssembly and Web Interface

To make the algorithm accessible to end users, the software was integrated into a web system equipped with tools for interacting with it. After implementing the algorithm in C++, it was compiled into WebAssembly [W3C24], a web standard that defines a binary and textual format for executing code within web pages at near-native speed. WebAssembly allows the algorithm to run efficiently in a browser, enabling users to interact with it without needing to install additional software. During the WebAssembly compilation, it is possible to specify which functions from the source code are exported for later invocation. In the C++ code, an instance of the *OneStackLayoutsEnumerator* class is initialized within the *main* function, and another function is defined to trigger the generation and output of the next 1-stack layout. This function was exported to WebAssembly, making it callable from the web system.

When compiling C++ into WebAssembly, a JavaScript script can be generated to manage the interaction with the WebAssembly executable. This script was modified to enable communication between the web interface and WebAssembly. Users can upload a file containing the graph structure, which is passed as a parameter to the *main* function, triggered by a button press. The script processes the algorithm's output, passing the vertex ordering for the generated 1-stack layout to a JavaScript function. Using the Vis Network library [Vis24], this function visualizes the corresponding 1-page book embedding. Additionally, the total number of 1-page layouts is output by the *main* function, retrieved by the script and displayed on the screen. Users can generate the next 1-stack layout by pressing a button, which invokes the exported function of the executable responsible for generating the next 1-stack layout. The generated layouts are also stored by the script, allowing the user to navigate through both previous and subsequent layouts.

Furthermore, the web interface allows users to explore the generated layouts through zoom and pan features, enhancing interactivity. The users can also download the image

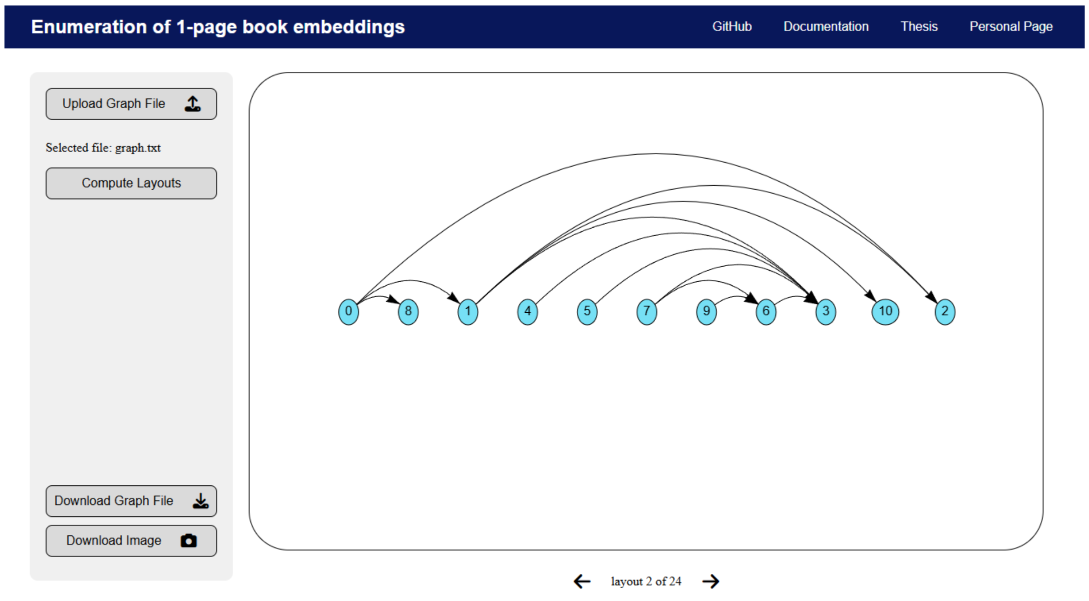

Figure 4.1: The web interface for enumerating 1-stack layouts.

of the current layout as well as the input file that generated the graph being displayed. Notably, this web system does not require a backend to function, as it relies solely on JavaScript to manage the input and output communication with the WebAssembly executable, which is one of its main advantages.

### 4.2.1 Encountered Challenges

During the development, several issues arose, particularly when compiling OGDF for WebAssembly. Certain assertions within the OGDF library failed in the WebAssembly environment, leading to unexpected runtime errors. One of the main challenges involved accessing and traversing the adjacency structure of node objects. OGDF's built-in methods for visiting node adjacencies were not functioning as expected, complicating the handling of graph connectivity.

To address these issues, supplementary data structures were introduced specifically for the WebAssembly version. These structures were used to track adjacency information and ensure the proper functionality of the graph traversal processes. Although this solution added some complexity, it enabled the algorithm to run smoothly within the web environment.

## 4.3 API and Interface

A custom library was developed to facilitate the enumeration of 1-stack layouts, which users can interact with through the web interface. The key class for enumeration is initialized with an instance of the *Graph* class from OGDF and provides the following member functions:

- *hasNext()*: Returns a boolean indicating whether more 1-stack layouts can be generated.
- *getNext()*: Returns the next valid 1-stack layout in the enumeration.
- *numberOfLayouts()*: Returns the total number of possible 1-stack layouts for the given DAG.

These functions allow users to interact with the algorithm and explore the different possible 1-stack layouts of their input DAGs. The web interface is designed to be intuitive, with clear options for stepping through the layouts and visualizing each configuration.

# Conclusions and Future Developments

In this thesis, 1-page book embeddings of DAGs have been explored. The first part described a known algorithm by Heath and Pemmaraju, for recognizing whether a given DAG admits a 1-stack layout. Building on this, an efficient algorithm for enumerating all possible 1-stack layouts of an $n$-vertex DAG has been presented, achieving a delay of $O(n)$ between successive layouts. The algorithm leverages the structure of the block-cutpoint tree to identify permutable blocks and generate all possible BFS orderings of the blocks that lead to the generation of distinct 1-stack layouts. The formal correctness of the algorithm has been demonstrated throughout the thesis. Additionally, a formula to count the total number of possible 1-stack layouts for a given DAG has been derived.

A practical outcome of this work is the implementation of the proposed algorithm in a web-based system. This system, built using WebAssembly, allows users to interactively explore the different possible 1-stack layouts of a DAG uploaded by the user, making the tool accessible without requiring any additional software installations.

## Theoretical Future Developments

From a theoretical perspective, future developments could extend to the enumeration of 1-stack layouts for non-connected DAGs. To achieve this, it would be necessary to enumerate the 1-stack layouts of all connected components of the DAG, which would then need to be combined through permutations and nesting, resulting in the generation of a large number of possible 1-stack layouts.

Another potential development involves summarizing all possible 1-stack layouts using data structures such as PQ-trees [BL76]. This may be achieved by using specialized PQ-trees, one for each rooting of the block-cutpoint tree identified by the algorithm, where permutable blocks are associated with P-nodes and non-permutable blocks are associated with non-flippable Q-nodes. In this way, a compact output may be obtained to represent all possible 1-stack layouts generated by the algorithm from a single rooting of the block-cutpoint tree.

## Practical Future Developments

From a practical perspective, future developments could lead to improvements in the visualization of 1-stack layouts within the web system. One planned enhancement involves adding a visualization of the block-cutpoint tree that leads to the generation of the current 1-stack layout. A possible new interaction introduced by such an improvement may consist in the ability to easily select blocks or subtrees of the block-cutpoint tree to be highlighted in the 1-page layout by directly interacting with the representation of the block-cutpoint tree.

Another potential development involves allowing users to specify certain constraints on the vertex positioning in the final 1-stack layout. For instance, a user might need a specific vertex of the input DAG to appear before another vertex in the layout. In this case, the user could add this constraint, and the system would display only the 1-stack layouts that satisfy such a condition, if any exist.

# Bibliography

[BDF$^+$23] Michael A. Bekos, Giordano Da Lozzo, Fabrizio Frati, Martin Gronemann, Tamara Mchedlidze, and Chrysanthi N. Raftopoulou. Recognizing dags with page-number 2 is np-complete. *Theor. Comput. Sci.*, 946:113689, 2023.

[BDG$^+$23] Carla Binucci, Giordano Da Lozzo, Emilio Di Giacomo, Walter Didimo, Tamara Mchedlidze, and Maurizio Patrignani. Upward book embeddability of st-graphs: Complexity and algorithms. *Algorithmica*, 85(12):3521–3571, 2023.

[BK79] Frank Bernhart and Paul C. Kainen. The book thickness of a graph. *Journal of Combinatorial Theory, Series B*, 27(3):320–331, 1979.

[BL76] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976.

[CGJ$^+$14] Michael Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 17, pages 543–569. CRC Press, 2014.

[DDF$^+$24] Giordano Da Lozzo, Giuseppe Di Battista, Fabrizio Frati, Fabrizio Grosso, and Maurizio Patrignani. Efficient enumeration of drawings and combinatorial structures for maximal planar graphs. In Ryuhei Uehara, Katsuhisa Yamanaka, and Hsu-Chun Yen, editors, *WALCOM: Algorithms and Computation*, pages 350–364, Singapore, 2024. Springer Nature Singapore.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing.* Prentice Hall, 1999.

[HP66] Frank Harary and Geert Prins. The block-cutpoint-tree of a graph. *Publicationes Mathematicae Debrecen*, 13:103–107, 1966.

[HP73] F. Harary and E.M. Palmer. *Graphical Enumeration.* Academic Press, 1973.

[HP99] Lenwood S. Heath and Sriram V. Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part ii. *SIAM Journal on Computing*, 28(5):1588–1626, 1999.

[HT73a] John E. Hopcroft and Robert E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[HT73b] John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973.

[Knu06] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming).* Addison-Wesley Professional, 2006.

[Knu11] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*, volume 4A. Addison-Wesley Professional, 1st edition, 2011.

[KW01] M. Kaufmann and D. Wagner. *Drawing Graphs: Methods and Models.* Springer, 2001.

[Vis24] Vis.js. Vis network documentation. `https://visjs.org/`, 2024. Accessed: 2024-10-02.

[W3C24] W3C WebAssembly Working Group. Webassembly core specification. `https://webassembly.github.io/spec/core/`, 2024. Accessed: 2024-10-02.

[Wes00] Douglas B. West. *Introduction to Graph Theory.* Prentice Hall, 2 edition, 2000.