Efficiently Listing Combinatorial Patterns in Graphs

Rui Ferreira

Supervisor
Roberto Grossi

Supervisor
Romeo Rizzi

Referee
Takeaki Uno

Referee
Stéphane Vialette

May 2013

Direttore della Scuola: Pierpaolo Degano

SSD: INF/01 - Informatica
First and foremost, a heartfelt thank you to my advisers Roberto Grossi and Romeo Rizzi – this thesis would not have been possible without your deep insights, support and encouragement. My gratitude to my co-authors Etienne Birmelé, Pierluigi Crescenzi, Roberto Grossi, Vicent Lacroix, Andrea Marino, Nadia Pisanti, Romeo Rizzi, Gustavo Sacomoto and Marie-France Sagot. I would also like to thank everyone at the Department of Computer Science of the University of Pisa, who made me feel at home in a distant country. It has been a pleasure working surrounded by these great people.

My deepest gratitude to my parents Fernando and Cila for their encouragement and unconditional love. A big hug to my sister Ana, brother-in-law Paulo, Andre & Miguel. To Hugo, João, Tina, Ernesto and all my family. – I cannot express how sorry I am for being away from you all.

Last but not least, to Desi for being by my side.

— Rui
Abstract

Graphs are extremely versatile and ubiquitous mathematical structures with potential to model a wide range of domains. For this reason, graph problems have been of interest since the early days of computer science. Some of these problems consider substructures of a graph that have certain properties. These substructures of interest, generally called patterns, are often meaningful in the domain being modeled. Classic examples of patterns include spanning trees, cycles and subgraphs.

This thesis focuses on the topic of explicitly listing all the patterns existing in an input graph. One of the defining features of this problem is that the number of patterns is frequently exponential on the size of the input graph. Thus, the time complexity of listing algorithms is parameterized by the size of the output.

The main contribution of this work is the presentation of optimal algorithms for four different problems of listing patterns in graphs. These algorithms are framed within the same generic approach, based in a recursive partition of the search space that divides the problem into subproblems. The key to an efficient implementation of this approach is to avoid recursing into subproblems that do not list any patterns. With this goal in sight, a dynamic data structure, called the certificate, is introduced and maintained throughout the recursion. Moreover, properties of the recursion tree and lower bounds on the number of patterns are used to amortize the cost of the algorithm on the size of the output.

The first problem introduced is the listing of all $k$-subtrees: trees of fixed size $k$ that are subgraphs of an undirected input graph. The solution is presented incrementally to illustrate the generic approach until an optimal output-sensitive algorithm is reached. This algorithm is optimal in the sense that it takes time proportional to the time necessarily required to read the input and write the output.

The second problem is that of listing $k$-subgraphs: connected induced subgraphs of size $k$ in an undirected input graph. An optimal algorithm is presented, taking time proportional to the size of the input graph plus the edges in the $k$-subgraphs.

The third and fourth problems are the listing of cycles and listing of paths between two vertices in an undirected input graph. An optimality-preserving reduction from listing cycles to listing paths is presented. Both problems are solved optimally, in time proportional to the size of the input plus the size of the output.

The algorithms presented improve previously known solutions and achieve optimal time bounds. The thesis concludes by pointing future directions for the topic.
Contents

1 Introduction 11
  1.1 Contribution .............................................. 12
    1.1.1 Listing k-subtrees ................................. 12
    1.1.2 Listing k-subgraphs ............................... 13
    1.1.3 Listing cycles and st-paths ....................... 13
  1.2 Organization ............................................ 14

2 Background 15
  2.1 Graphs ................................................... 15
    2.1.1 Applications ........................................ 15
    2.1.2 Definitions ......................................... 16
  2.2 Combinatorial patterns in graphs ......................... 17
    2.2.1 k-subtrees .......................................... 17
    2.2.2 k-subgraphs ........................................ 18
    2.2.3 st-paths and cycles ................................ 19
    2.2.4 Induced paths, chordless cycles and holes ....... 19
    2.2.5 st-bubbles .......................................... 20
  2.3 Listing .................................................. 20
    2.3.1 Complexity .......................................... 20
    2.3.2 Techniques ........................................ 21
    2.3.3 Problems ........................................... 22
  2.4 Overview of the techniques used .......................... 22
    2.4.1 Binary partition method ............................ 23
    2.4.2 Certificates ......................................... 23
    2.4.3 Adaptive amortized analysis ....................... 23

3 Listing k-subtrees 25
  3.1 Preliminaries ........................................... 26
  3.2 Basic approach: recursion tree ......................... 27
    3.2.1 Top level ........................................... 27
    3.2.2 Recursion tree and analysis ....................... 28
  3.3 Improved approach: certificates ......................... 29
    3.3.1 Introducing certificates .......................... 29
# CHAPTER 0. CONTENTS

| Section | Page |
|---------|------|
| 3.3.2 Maintaining the invariant | 34 |
| 3.3.3 Analysis | 38 |
| 3.4 Optimal approach: amortization | 38 |
| 3.4.1 Internal edges | 39 |
| 3.4.2 Amortization | 40 |
| 4 Listing k-subgraphs | 45 |
| 4.1 Preliminaries | 47 |
| 4.2 Top-level algorithm | 47 |
| 4.3 Recursion | 48 |
| 4.4 Amortization strategy | 51 |
| 4.5 Certificate | 51 |
| 4.5.1 Maintaining the certificate | 53 |
| 4.6 Other operations | 57 |
| 5 Listing cycles and st-paths | 61 |
| 5.1 Preliminaries | 64 |
| 5.2 Overview and main ideas | 65 |
| 5.2.1 Reduction to st-paths | 65 |
| 5.2.2 Decomposition in biconnected components | 66 |
| 5.2.3 Binary partition scheme | 67 |
| 5.2.4 Introducing the certificate | 68 |
| 5.2.5 Recursion tree and cost amortization | 70 |
| 5.3 Amortization strategy | 72 |
| 5.4 Certificate implementation and maintenance | 74 |
| 5.5 Extended analysis of operations | 78 |
| 5.5.1 Operation right_update(C,e) | 79 |
| 5.5.2 Operation left_update(C,e) | 81 |
| 6 Conclusion and future work | 85 |
| Bibliography | 87 |
# List of Figures

2.1 Examples of graphs ................................................................. 16  
2.2 Examples of patterns .............................................................. 18  
3.1 Example graph $G_1$ and its 3-trees ............................................. 26  
3.2 Recursion tree of $\text{ListTrees}_v$ for graph $G_1$ in Fig. 2.2(a) and $v_i = a$. 30  
3.3 Choosing edge $e \in C(S)$. The certificate $D$ is shadowed. ................. 35  
4.1 Example graph $G_1$ and its 3-subgraphs ....................................... 46  
4.2 Choosing vertex $v \in N^*(S)$. The certificate $C$ is shadowed. ............... 53  
4.3 Case (a) of $\text{update}_\text{right}(C, v)$ when $v$ is internal .................... 55  
4.4 Case (b) of $\text{update}_\text{right}(C, v)$ when $v$ is internal .................... 56  
5.1 Example graph $G_1$, its st-paths and cycles .................................. 62  
5.2 Diamond graph. ................................................................. 63  
5.3 Block tree of $G$ with bead string $B_{s,t}$ in gray. .............................. 64  
5.4 Example certificate of $B_{u,t}$ .................................................... 68  
5.5 Spine of the recursion tree ..................................................... 71  
5.6 Application of $\text{right_update}(C, e)$ on a spine of the recursion tree .. 79  
5.7 Block tree after removing vertex $u$ ............................................ 82  
5.8 Certificates of the left branches of a spine .................................... 84
Chapter 1

Introduction

This chapter presents the contributions and organization of this thesis. Let us start by briefly framing the title "Efficiently Listing Combinatorial Patterns in Graphs".

**Graphs.** Graphs are an ubiquitous abstract model of both natural and man-made structures. Since Leonhard Euler’s famous use of graphs to solve the Seven Bridges of Königsberg [19, 33], graph models have been applied in computer science, linguistics, chemistry, physics and biology [14, 70]. Graphs, being discrete structures, are posed for problems of combinatorial nature [14] – often easy to state and difficult to solve.

**Combinatorial patterns.** The term “pattern” is used as an umbrella of concepts to describe substructures and attributes that are considered to have significance in the domain being modeled. Searching, matching, enumerating and indexing patterns in strings, trees, regular expressions or graphs are widely researched areas of computer science and mathematics. This thesis is focused on patterns of combinatorial nature. In the particular case of graphs, these patterns are subgraphs or substructures that have certain properties. Examples of combinatorial patterns in graphs include spanning trees, graph minors, subgraphs and paths [49, 76].

**Listing.** The problem of graph enumeration, pioneered by Pólya, Caley and Redfield, focuses on counting the number of graphs with certain properties as a function of the number of vertices and edges in those graphs [38]. Philippe Flajolet and Robert Sedgewick have introduced techniques for deriving generating functions of such objects [23]. The problem of counting patterns occurring in graphs (as opposed to the enumeration of the graphs themselves) is more algorithmic in nature. With the introduction of new theoretical tools, such as parameterized complexity theory [17, 24], several new results have been achieved in the last decade [34, 51].

In the research presented, we tackle the closely related problem of listing patterns in graphs: i.e. explicitly outputting each pattern found in an input graph. Although this problem can be seen as a type of exhaustive enumeration (in fact, it is common in the literature that the term enumeration is used), the nature of both problems is considered different as the patterns have to be explicitly generated [49, 76].
CHAPTER 1. INTRODUCTION

Complexity. One of the defining features of the problem of listing combinatorial patterns is that there frequently exists an exponential number of patterns in the input graph. This implies that there are no polynomial-time algorithms for this family of problems. Nevertheless, the time complexity of algorithms for the problem of listing patterns in graphs can still be analyzed. The two most common approaches are: (i) output-sensitive analysis, i.e. analyzing the time complexity of the algorithm as a function of the output and input size, and (ii) time delay, i.e. bounding the time it takes from the output of one pattern to the next in function of the size of the input graph and the pattern.

1.1 Contribution

This thesis presents a new approach for the problem of listing combinatorial patterns in an input graph $G$ with uniquely labeled vertices. At the basis of our technique [20][10], we list the set of patterns in $G$ by recursively using a binary partition: (i) listing the set of patterns that include a vertex (or edge), and (ii) listing the set of patterns that do not include that vertex (resp. edge).

The core of the technique is to maintain a dynamic data structure that efficiently tests if the set of patterns is empty, avoiding branches of the recursion that do not output any patterns. This problem of dynamic data structures is very different from the classical view of fully-dynamic or decrementally-dynamic data structures. Traditionally, the cost of operations in a dynamic data structure is amortized over a sequence of arbitrary operations. In our case, the binary-partition method induces a well defined order in which the operations are performed. This allows a much better amortization of the cost. Moreover, the existence of lower bounds (even if very weak) on the number of the combinatorial patterns in a graph allows a better amortization of the cost of the recursion and the maintenance of the dynamic data structure.

This approach is applied to the listing of the following four different patterns in undirected graphs: $k$-subtrees, $k$-subgraphs, $st$-paths and cycles. In all four cases, we obtain optimal output-sensitive algorithms, running in time proportional to the size of the input graph plus the size of the output patterns.

1.1.1 Listing $k$-subtrees

Given an undirected connected graph $G = (V, E)$, where $V$ is the set of vertices and $E$ the set of edges, a $k$-subtree $T$ is a set of edges $T \subseteq E$ that is acyclic, connected and contains $k$ vertices. We present the first optimal output-sensitive algorithm for listing the $k$-subtrees in input graph $G$. When $s$ is the number of $k$-subtrees in $G$, our algorithm solves the problem in $O(sk)$ time.

---

1This result has been published in [20].
2These results have been published in [10].
We present our solution starting from the binary-partition method. We divide the problem of listing the \( k \)-subtrees in \( G \) into two subproblems by taking an edge \( e \in E \): (i) we list the \( k \)-subtrees that contain \( e \), and (ii) list those that do not contain \( e \). We proceed recursively on these subproblems until there is just one \( k \)-subtree to be listed. This method induces a binary recursion tree, and all the \( k \)-subtrees are listed when reaching the leaves of this recursion tree.

Although this first approach is output sensitive, it is not optimal. One problem is that each adjacency list in \( G \) can be of length \( O(n) \), but we cannot pay such a cost in each recursive call. Also, we need to maintain a certificate throughout the recursive calls to guarantee a priori that at least one \( k \)-subtree will be generated. By exploiting more refined structural properties of the recursion tree, we present our algorithmic ideas until an optimal output-sensitive listing is obtained.

### 1.1.2 Listing \( k \)-subgraphs

When considering an undirected connected graph \( G \), a \( k \)-subgraph is a connected subgraph of \( G \) induced by a set of \( k \) vertices. We propose an algorithm to solve the problem of listing all the \( k \)-subgraphs in \( G \). Our algorithm is optimal: solving the problem in time proportional to the size of the input graph \( G \) plus the size of the edges in the \( k \)-subgraphs to output.

We apply the binary-partition method, dividing the problem of listing all the \( k \)-subgraphs in two subproblems by taking a vertex \( v \in V \): we list the \( k \)-subgraphs that contain \( v \); and those that do not contain \( v \). We recurse on these subproblems until no \( k \)-subgraphs remain to be listed. This method induces a binary recursion tree where the leaves correspond to \( k \)-subgraphs.

In order to reach an optimal algorithm, we maintain a certificate that allows us to determine efficiently if there exists at least one \( k \)-subgraph in \( G \) at any point in the recursion. Furthermore, we select the vertex \( v \) (which divides the problem into two subproblems), in a way that facilitates the maintenance of the certificate.

### 1.1.3 Listing cycles and \( st \)-paths

Listing all the simple cycles (hereafter just called cycles) in a graph is a classical problem whose efficient solutions date back to the early 70s. For a directed graph with \( n \) vertices and \( m \) edges, containing \( \eta \) cycles, the best known solution in the literature is given by Johnson’s algorithm \[43\] and takes \( O((\eta + 1)(m + n)) \) time. This algorithm can be adapted to undirected graphs, maintaining the same time complexity. Surprisingly, this time complexity is not optimal for undirected graphs: to the best of our knowledge, no theoretically faster solutions have been proposed in almost 40 years.

We present the first optimal solution to list all the cycles in an undirected graph \( G \), improving the time bound of Johnson’s algorithm. Specifically, let \( \mathcal{C}(G) \) denote the set of all these cycles, and observe that \( |\mathcal{C}(G)| = \eta \). For a cycle \( c \in \mathcal{C}(G) \), let \( |c| \)
denote the number of edges in \( c \). Our algorithm requires \( O(m + \sum_{c \in C(G)} |c|) \) time and is asymptotically optimal: indeed, \( \Omega(m) \) time is necessarily required to read the input \( G \), and \( \Omega(\sum_{c \in C(G)} |c|) \) time is required to list the output.

We also present the first optimal solution to list all the simple paths from \( s \) to \( t \) (shortly, \( st \)-paths) in an undirected graph \( G \). Let \( P_{st}(G) \) denote the set of \( st \)-paths in \( G \) and, for an \( st \)-path \( \pi \in P_{st}(G) \), let \( |\pi| \) be the number of edges in \( \pi \). Our algorithm lists all the \( st \)-paths in \( G \) optimally in \( O(m + \sum_{\pi \in P_{st}(G)} |\pi|) \) time, observing that \( \Omega(\sum_{\pi \in P_{st}(G)} |\pi|) \) time is required to list the output.

While the basic approach is simple, we use a number of non-trivial ideas to obtain our optimal algorithm for an undirected graph \( G \). We start by presenting an optimality-preserving reduction from the problem of listing cycles to the problem of listing \( st \)-paths. Focusing on listing \( st \)-paths, we consider the decomposition of the graph into biconnected components and use the property that \( st \)-paths pass in certain articulation points to restrict the problem to one biconnected component at a time. We then use the binary-partition method to list: (i) \( st \)-paths that contain an edge \( e \), and (ii) those that do not contain \( e \). We make use of a certificate of existence of at least one \( st \)-path and prove that the cost of maintaining the certificate throughout the recursion can be amortized. A key factor of the amortization is the existence of a lower bound on the number of \( st \)-paths in a biconnected component.

### 1.2 Organization

After this brief introduction, let us outline the organization of this thesis.

Chapter 2 introduces background information on the topic at hand. In Section 2.1 we start by illustrating uses of graphs and reviewing basic concepts in graph theory. Section 2.2 motivates the results presented, by introducing combinatorial patterns and their applications in diverse areas. We proceed into Section 2.3 where an overview of listing and enumeration of patterns is given and the state of the art is reviewed. For a review of the state of the art on each problem tackled, the reader is referred to the introductory text of the respective chapter. We finalize the chapter with Section 2.4 which includes an overview of the techniques used throughout the thesis and explains how they are framed within the state of the art.

Chapters 3 to 5 give a detailed and formal view of the problems handled and results achieved. In Chapter 3 we provide an incremental introduction to the optimal output-sensitive algorithm to list \( k \)-subtrees, starting from a vanilla version of the binary-partition method and step by step introducing the certificate and amortization techniques used. Chapter 4 directly presents the optimal output-sensitive algorithm for the problem of listing \( k \)-subgraphs. Similarly, the results achieved on the problems of listing cycles and \( st \)-paths are presented in Chapter 5.

We finalize this exposition with Chapter 6 reviewing the main contributions and presenting some future directions and improvements to the efficient listing of combinatorial patterns in graphs.
Chapter 2

Background

2.1 Graphs

Graphs are abstract mathematical structures that consist of objects, called vertices, and links that connect pairs of vertices, called edges. Although the term graph was only coined in 1876 by Sylvester, the study of graphs dates back to 1736 when Euler published his famous paper on the bridges of Königsberg [33, 19]. With the contributions of Caley, Pólya, De Bruijn, Peterson, Robertson, Seymour, Erdős, Rényi and many many others, the field of graph theory developed and provided the tools for what is considered one of the most versatile mathematical structures.

A classical example of a graph is the rail network of a country: vertices represent the stations and there is an edge between two vertices if they represent consecutive stations along the same railroad. This is an example of an undirected graph since the notion of consecutive station is a symmetric relation. Another example is provided by the World Wide Web: the vertices are the websites and two are adjacent if there is a direct link from one to the other. Graphs of this latter type are called directed graphs, since a link from one website to the other does not imply the reverse. These edges are sometimes called directed edges or arcs.

2.1.1 Applications

There are vast practical uses of graphs. As an example, Stanford Large Dataset Collection [54] includes graphs from domains raging from jazz musicians to metabolic networks. In this section, we take a particular interest in graphs as a model of biological functions and processes. For further information about different domains, [70, 14] are recommended as a starting point.

Metabolic networks. The physiology of living cells is controlled by chemical reactions and regulatory interactions. Metabolic pathways model these individual processes. The collection of distinct pathways co-existing within a cell is called a metabolic network. Interestingly, with the development of the technology to se-
sequence the genome, it has been possible to link some of its regions to metabolic pathways. This allows a better modeling of these networks and, through simulation and reconstruction, it possible to have in-depth insight into the key features of the metabolism of organisms.

**Protein-protein interaction networks.** Proteins have many biological functions, taking part in processes such as DNA replication and mediating signals from the outside to the inside of a cell. In these processes, two or more proteins bind together. These interactions play important roles in diseases (e.g. cancer) and can be modeled through graphs.

**Gene regulatory networks.** Segments of DNA present in a cell interact with each other through their RNA and protein expression products. There are also interactions occurring with the other substances present in the cell. These processes govern the rates of transcription of genes and can be modeled through a network.

**Signaling networks.** These networks integrate metabolic, protein-protein interaction and gene regulatory networks to model how signals are transduced within or between cells.

### 2.1.2 Definitions

Let us now introduce some formalism and notation when dealing with graphs. We will recall some of the concepts introduced here on the preliminaries of Chapters 3-5.

**Undirected graphs.** A graph $G$ is an ordered pair $G = (V, E)$ where $V$ is the set of vertices and $E$ is the set of edges. The *order* of a graph is the number of vertices $|V|$ and its *size* the number of edges $|E|$. In the case of undirected graphs, an edge $e \in E$ is an unordered pair $e = (u, v)$ with $u, v \in V$. Graph $G$ is said to be simple if: (i) there are no *loops*, i.e. edges that start and end in the same vertex, and (ii) there are no *parallel edges*, i.e. multiple edges between the same pair of vertices. Otherwise, $E$ is a multiset and $G$ is called a *multigraph*. Figure 2.1(a) shows the example of undirected graph $G_1$.

Given a simple undirected graph $G = (V, E)$, vertices $u, v \in V$ are said to be *adjacent* if there exists an edge $(u, v) \in E$. The set of vertices adjacent to a vertex

![Figure 2.1: Examples of graphs](image-url)
2.2. COMBINATORIAL PATTERNS IN GRAPHS

$u$ is called its \textit{neighborhood} and is denoted by \( N(u) = \{v \mid (u, v) \in E \} \). An edge \( e \in E \) is incident in $u$ if $u \in e$. The \textit{degree} \( \text{deg}(u) \) of a vertex $u \in V$ is the number of edges incident in $u$, that is \( \text{deg}(u) = |N(u)| \).

**Directed graphs.** In case of \textit{directed graph} \( G = (V, E) \), an edge $e \in E$ has an orientation and therefore is an \textit{ordered} pair of vertices $e = (u, v)$. \( G_2 \) shows the example of directed graph \( G \). The neighborhood $N(u)$ of a vertex $u \in V$, is the union of the \textit{out-neighborhood} $N^+(u) = \{v \mid (u, v) \in E \}$ and \textit{in-neighborhood} $N^−(u) = \{v \mid (v, u) \in E \}$. Likewise, the degree $\text{deg}(u)$ is the union of the \textit{out-degree} $\text{deg}^+(u) = |N^+(u)|$ and the \textit{in-degree} $\text{deg}^−(u) = |N^−(u)|$.

**Subgraphs.** A graph $G' = (V', E')$ is said to be a \textit{subgraph} of $G = (E, V)$ if $V' \subseteq V$ and $E' \subseteq E$. The subgraph $G'$ is said to be \textit{induced} if and only if $e \in E'$ for every edge $e = (u, v) \in E$ with $u, v \in V'$. A subgraph of $G$ induced by a vertex set $V'$ is denoted by $G[V']$.

**Biconnected graphs.** An undirected graph $G = (V, E)$ is said to be \textit{biconnected} if it remains connected after the removal of any vertex $v \in V$. The complete graph of two vertices if considered biconnected. \( G_3 \) shows the example of biconnected graph $G$. A \textit{biconnected component} is a maximal biconnected subgraph. An \textit{articulation point} (or \textit{cut vertex}) is any vertex whose removal increases the number of biconnected components in $G$. Note that any connected graph can be decomposed into a tree of biconnected components, called the \textit{block tree} of the graph. The biconnected components in the block tree are attached to each other by shared articulation points.

# 2.2 Combinatorial patterns in graphs

Combinatorial patterns in an input graph are constrained substructures of interest for the domain being modeled. As an example, a cycle in a graph modeling the railway network of a country could represent a service that can be efficiently performed by a single train. In different domains, there exists a myriad of different structures with meaningful interpretations. We focus on generic patterns such as subgraphs, trees, cycles and paths, which have broad applications in different domains. In addition to the patterns studied in this thesis, we also introduce other patterns where the techniques presented are likely to have applications.

## 2.2.1 $k$-subtrees

Given a simple (without loops or parallel edges), undirected and connected graph $G = (V, E)$ with $n = |V|$ and an integer $k \in [2, n]$, a \textit{$k$-subtree} $T$ of $G$ is an acyclic connected subgraph of $G$ with $k$ vertices. Formally, $T \subseteq E$, $|T| = k - 1$, and $T$ is an unrooted, unordered, free tree. \( T_1 \) of graph $G_1$ from Figure 2.1(a) shows the example of the 4-subtree $T_1$ of graph $G_1$ from Figure 2.1(a).
Trees that are a subgraph of an input graph $G = (V, E)$ have been considered significant since the early days of graph theory. Examples of such trees that have been deeply studied include spanning trees [32] and Steiner trees [42]. Interestingly, when $k = |V|$, the $k$-subtrees of $G$ are its spanning trees. Thus, $k$-subtrees can be considered a generalization of spanning trees.

In several domains, e.g. biological networks, researchers are interested in local structures [29, 12] that represent direct interactions of the components of the network (see Section 2.1.1). In these domains, $k$-subtrees model acyclic interactions between those components. In [102], the authors present related applications of $k$-subtrees.

### 2.2.2 $k$-subgraphs

Given an undirected graph $G = (V, E)$, a set of vertices $V' \subseteq V$ induces a subgraph $G[V'] = (V', E')$ where $E' = \{(u, v) \mid u, v \in V' \text{ and } (u, v) \in E\}$. A $k$-subgraph is a connected induced subgraph $G[V']$ with $k$ vertices. Figure 2.2(b) shows the example of the 4-subgraph $G_1[V']$ with $V' = \{b, c, d, e\}$ (graph $G_1$ is available on Figure 2.1(a)).
2.2. COMBINATORIAL PATTERNS IN GRAPHS

The subgraphs of an input graph have been subject of study of many researchers. We highlight the recent interest of the bioinformatics community in network motifs. Network motifs are k-subgraphs that appear more frequently in an input graph than what would be expected in random networks. These motifs are likely to be components in the function of the network. In fact, motifs extracted from gene regulatory networks have been shown to perform definite functional roles.

2.2.3 st-paths and cycles

Given a directed or undirected graph $G = (V,E)$, a path in $G$ is a subgraph $\pi = (V',E') \subseteq G$ of the form:

$$ V' = \{u_1,u_2,\ldots,u_k\} \quad E' = \{(u_1,u_2), (u_2,u_3), \ldots, (u_{k-1},u_k)\} $$

where all the $u_i \in V'$ are distinct (and therefore a path is simple by definition). We refer to a path $\pi$ by its natural sequence of vertices or edges. A path $\pi$ from $s$ to $t$, or $st$-path, is denoted by $\pi = s \leadsto t$. Figure 2.2(c) shows the example of the ab-path $\pi_1$ in graph $G_1$ from Figure 2.1(a).

When $\pi = u_1,u_2,\ldots,u_k$ is a path, $k \geq 3$ and edge $(u_k,u_1) \in E$ then $c = \pi + (u_k,u_1)$ is a cycle in $G$. Figure 2.2(d) shows the example of the cycle $c_1$ in graph $G_1$ from Figure 2.1(a). We denote the number of edges in a path $\pi$ by $|\pi|$ and in a cycle $c$ by $|c|$.

Cycles have broad applications in many fields. Following our bioinformatics theme, cycles in metabolic networks represent cyclic metabolic pathways which are known to often be functionally important (e.g. Krebs cycle). Other domains where cycles and st-paths are considered important include symbolic model checking, telecommunication networks and circuit design.

2.2.4 Induced paths, chordless cycles and holes

Given an undirected graph $G = (V,E)$ and vertices $s,t \in V$, an induced path $\pi = s \leadsto t$ is an induced subgraph of $G$ that is a path from $s$ to $t$. By definition, there exists an edge between each pair of vertices adjacent in $\pi$ and there are no edges that connect two non-adjacent vertices. Figure 2.2(e) shows the example of the induced path $\pi_2$ in graph $G_3$ from Figure 2.1(c). Note that $\pi_1$ in Figure 2.2(e) is not an induced path in $G_3$ due to the existence of edges $(a,b)$ and $(b,e)$.

Similarly, a chordless cycle $c$ is an induced subgraph of $G$ that is a cycle. Chordless cycles are also sometimes called induced cycles or, when $|c| > 4$, holes. Figure 2.2(f) shows the example of the chordless cycle $c_2$ in graph $G_3$ from Figure 2.1(c).

Many important graph families are characterized in terms of induced paths and induced cycles. One example are chordal graphs: graphs with no holes. Other examples include distance hereditary graphs, trivially perfect graphs and block graphs.
2.2.5 \textit{st}-bubbles

Given a directed graph $G = (V,E)$, and two vertices $s, t \in V$, a \textit{st}-\textit{bubble} $b$ is an unordered pair $b = (P,Q)$ of \textit{st}-paths $P,Q$ whose inner-vertices are disjoint (i.e.: $P \cap Q = \{s,t\}$). The term \textit{bubble} (also called a \textit{mouth}) refers to \textit{st}-bubbles without fixing the source and target (i.e. $\forall s,t \in V$). Figure 2.2(g) shows the example of the \textit{ac}-bubble $b_1$ in directed graph $G_2$ from Figure 2.1(b).

Bubbles represent polymorphisms in models of the DNA. Specifically, in De Bruijn graphs (a directed graph) generated by the reads of a DNA sequencing project, bubbles can represent two different traits occurring in the same species or population [69, 74]. Moreover, bubbles can can also represents sequencing errors in the DNA sequencing project [83, 109, 9].

2.3 Listing

Informally, given an input graph $G$ and a definition of pattern $P$, a listing problem asks to output all substructures of graph $G$ that satisfy the properties of pattern $P$.

Listing combinatorial patterns in graphs has been a long-time problem of interest. In his 1970 book [66], Moon remarks “Many papers have been written giving algorithms, of varying degrees of usefulness, for listing the spanning trees of a graph”\footnote{This citation was found in [76]}. Among others, he cites [35, 18, 101, 22, 21] – some of these early papers date back to the beginning of the XX century. More recently, in the 1960s, Minty proposed an algorithm to list all spanning trees [64]. Other results from Welch, Tiernan and Tarjan for this and other problems soon followed [104, 90, 89].

It is not easy to find a reference book or survey with the state of the art in this area. [38, 8, 76, 49] partially cover the material. In this section we present a brief overview of the theory, techniques and problems of the area. For a review of the state of the art related with the problems tackled, we refer the reader to the introductory text of each chapter.

2.3.1 Complexity

One defining characteristic of the problem of listing combinatorial patterns in graphs is that the number of patterns present in the input is often exponential in the input size. Thus, the number of patterns and the output size have to be taken into account when analyzing the time complexity of these algorithms. Several notions of output-sensitive efficiency have been proposed:

1. \textit{Polynomial total time}. Introduced by Tarjan in [89], a listing algorithm runs in polynomial total time if its time complexity is polynomial in the input and output size.
2. **P-enumerability.** Introduced by Valiant in [99, 100], an algorithm P-enumerates the patterns of a listing problem if its running time is $O(p(n)s)$, where $p(n)$ is a polynomial in the input size $n$ and $s$ is the number of solutions to output. When this algorithm uses space polynomial in the input size only, Fukuda introduced the term strong P-enumerability [25].

3. **Polynomial delay.** An algorithm has delay $D$ if: (i) the time taken to output the first solution is $O(D)$, and (ii) the time taken between the output of any two consecutive solutions is $O(D)$. Introduced by Johnson, Yannakakis and Papadimitriou [44], an algorithm has polynomial (resp. linear, quadratic, . . . ) delay if $D$ is polynomial (resp. linear, quadratic, . . . ) in the size of the input.

In [75], Rospocher proposed a hierarchy of complexity classes that take these concepts into account. He introduces a notion of reduction between these classes and some listing problems were proven to be complete for the class $L_P$: the listing analogue of class $\#P$ for counting problems.

We define an algorithm for a listing problem to be optimally output sensitive if the running time of the algorithm is $O(n + q)$, where $n$ is the input size and $q$ is the size of the output. Although this is a notion of optimality for when the output has to be explicitly represented, it is possible that the output can be encoded in the form of the differences between consecutive patterns in the sequence of patterns to be listed.

### 2.3.2 Techniques

Since combinatorial patterns in graphs have different properties and constraints, it is complex to have generic algorithmic techniques that work for large classes of problems. Some of the ideas used for the listing of combinatorial structures [30, 52, 107] can be adapted to the listing of combinatorial patterns in those structures. Let us present some of the known approaches.

**Backtrack search.** According to this approach, a backtracking algorithm finds the solutions for a listing problem by exploring the search space and abandoning a partial solution (thus, the name “backtracking”) that cannot be completed to a valid one. For further information see [73].

**Binary partition.** An algorithm that follows this approach recursively divides the search space into two parts. In the case of graphs, this is generally done by taking an edge (or a vertex) and: (i) searching for all solutions that include that edge (resp. vertex), and (ii) searching for all solutions that do not include that edge (resp. vertex). This method is presented with further detail in Section 2.4.1.

**Contraction–deletion.** Similarly to the binary-partition approach, this technique is characterized by recursively dividing the search space in two. By taking an edge of the graph, the search space is explored by: (i) contraction of the edge,
i.e. merging the endpoints of the edge and their adjacency list, and (ii) deletion of the edge. For further information the reader is referred to [13].

**Gray codes.** According to this technique, the space of solutions is encoded in such a way that consecutive solutions differ by a constant number of modifications. Although not every pattern has properties that allow such encoding, this technique leads to very efficient algorithms. For further information see [78].

**Reverse search** This is a general technique to explore the space of solutions by reversing a local search algorithm. Considering a problem with a unique maximum objective value, there exist local search algorithms that reach the objective value using simple operations. One such example is the Simplex algorithm. The idea of reverse search is to start from this objective value and reverse the direction of the search. This approach implicitly generates a tree of the search space that is traversed by the reverse search algorithm. One of the properties of this tree is that it has bounded height, a useful fact for proving the time complexity of the algorithm. Avis and Fukuda introduced this idea in [5].

Although there is some literature on techniques for enumeration problems [97, 98], many more techniques and “tricks” have been introduced when attacking particular problems. For a deep understanding of the topic, the reader is recommended to review the work of David Avis, Komei Fukuda, Shin-ichi Nakano, Takeaki Uno.

### 2.3.3 Problems

As a contribution for researchers starting to explore the topic of listing patterns in graphs, we present a table with the most relevant settings of problems and a list of state of the art references. For the problems tackled in this thesis, a more detailed review is presented on the introductory text of Chapters 3, 4 and 5.

| Problem                      | References                  |
|------------------------------|-----------------------------|
| Cycles and Paths             | See [10] and [43]           |
| Spanning trees               | See [82]                    |
| Subgraphs                    | See [5], [50]               |
| Matchings                    | See [93], [72], [26] and [94] |
| Cut-sets                     | See [91], [4]               |
| Independent sets             | See [44]                    |
| Induced paths, cycles, holes | See [95]                    |
| Cliques, pseudo-cliques     | See [65], [56] and [96]     |
| Colorings                    | See [59] and [58]           |

### 2.4 Overview of the techniques used

In this section, we present an overview of the approach we have applied to problems of listing combinatorial patterns in graphs. Let us start by describing the binary partition method and then introduce the concept of certificate.
2.4.1 Binary partition method

Binary partition is a method for recursively subdividing the search space of the problem. In the case of a graph \( G = (V, E) \), we can take an edge \( e \in E \) (or a vertex \( v \in V \)) and list: (i) all the patterns that include \( e \), and (ii) all the patterns that do not include edge \( e \).

Formally, let \( S(G) \) denote the set of patterns in \( G \). For each pattern \( p \in S(G) \), a \textit{prefix} \( p' \) of \( p \) is a substructure of pattern \( p \), denoted by \( p' \subseteq p \). Let \( S(p', G) \) be the set of patterns in \( G \) that have \( p' \) as a substructure. By taking an edge \( e \in E \), we can write the following recurrence:

\[
S(p', G) = S(p' + \{e\}, G) \cup S(p', G - \{e\}) \quad (2.1)
\]

Noting that the left side is disjoint with the right side of the recurrence, this can be a good starting point to enumerate the patterns in \( G \).

In order to implement this idea efficiently, one key point is to maintain \( S(p', G) \neq \emptyset \) as an invariant. Generally speaking, maintaining the invariant on the left side of the recurrence is easier than on the right side. Since we can take any edge \( e \in E \) to partition the space, we can use the definition of the pattern to select an edge that maintains the invariant. For example, if the pattern is connected, we augment the prefix with an edge that is connected to it. In order to maintain this invariant efficiently, we introduce the concept of a certificate.

2.4.2 Certificates

When implementing an algorithm using recurrence \([2.1]\) we maintain a certificate that guarantees that \( S(p', G) \neq \emptyset \). An example of a certificate is a pattern \( p \supseteq p' \). In the recursive step, we select an edge \( e \) that interacts as little as possible with the certificate and thus facilitates its maintenance. In the ideal case, a certificate \( C \) is both the certificate of \( S(p', G) \) and of \( S(p', G - \{e\}) \). Although this is not always possible, we are able to amortize the cost of modifying the certificate.

During the recursion we are able to avoid copying the certificate and maintain it instead. We use data structures that preserve previous versions of itself when they are modified. These data structures, usually called \textit{persistent}, allow us to be efficient in terms of the space.

2.4.3 Adaptive amortized analysis

The problem of maintaining a certificate is very different than the traditional view of dynamic data structures. When considering fully-dynamic or partially-dynamic data structures, the cost of operations is usually amortized after a sequence of \textit{arbitrary} insertions and deletions. In the case of maintaining the certificate throughout the recursion, the sequence of operations is not arbitrary. As an example, when we
remove an edge from the graph, the edge is added back again on the callback of the recursion. Furthermore, we have some control over the recursion as we can select the edge $e$ in a way that facilitates the maintenance of the certificate (and its analysis).

Another key idea is that a certificate can provide a lower bound on the size of set $S(p', G)$. We have designed certificates whose time to maintain is proportional to the lower bound on the number of patterns they provide.

Summing up, the body of work presented in this thesis revolves around shaping the recursion tree, designing and efficiently implementing the certificate while using graph theoretical properties to amortize the costs of it all.
Chapter 3

Listing k-subtrees

Given an undirected connected graph \( G = (V, E) \) with \( n \) vertices and \( m \) edges, we consider the problem of listing all the \( k \)-subtrees in \( G \). Recall that we define a \( k \)-subtree \( T \) as an edge subset \( T \subseteq E \) that is acyclic and connected, and contains \( k \) vertices. Refer to Section 2.2.1 for applications of this combinatorial pattern. We denote by \( s \) the number of \( k \)-subtrees in \( G \). For example, there are \( s = 9 \) \( k \)-subtrees in the graph of Fig. 3.1, where \( k = 3 \). Originally published in [20], we present the first optimal output-sensitive algorithm for listing all the \( k \)-subtrees in \( O(sk) \) time, using \( O(m) \) space.

As a special case, this problem also models the classical problem of listing the spanning trees in \( G \), which has been largely investigated (here \( k = n \) and \( s \) is the number of spanning trees in \( G \)). The first algorithmic solutions appeared in the 60’s [64], and the combinatorial papers even much earlier [66]. Read and Tarjan presented an output-sensitive algorithm in \( O(sm) \) time and \( O(m) \) space [73]. Gabow and Myers proposed the first algorithm [27] which is optimal when the spanning trees are explicitly listed. When the spanning trees are implicitly enumerated, Kapoor and Ramesh [46] showed that an elegant incremental representation is possible by storing just the \( O(1) \) information needed to reconstruct a spanning tree from the previously enumerated one, giving \( O(m+s) \) time and \( O(mn) \) space [46], later reduced to \( O(m) \) space by Shioura et al. [82]. After we introduced the problem in [20], a constant-time algorithm for the enumeration of \( k \)-subtrees in the particular case of trees [102] was introduced by Wasa, Uno and Arimura. We are not aware of any other non-trivial output-sensitive solution for the problem of listing the \( k \)-subtrees in the general case.

We present our solution starting from the binary partition method (Section 2.4.1). We divide the problem of listing all the \( k \)-subtrees in two subproblems by choosing an edge \( e \in E \): we list the \( k \)-subtrees that contain \( e \) and those that do not contain \( e \). We proceed recursively on these subproblems until there is just one \( k \)-subtree to be listed. This method induces a binary recursion tree, and all the \( k \)-subtrees can be listed when reaching the leaves of the recursion tree.
26

CHAPTER 3. LISTING K-SUBTREES

(a) Graph $G_1$  
(b) 3-trees of $G_1$ containing vertex $a$

(c) 3-trees of $G_1 - \{a\}$ containing vertex $b$
(d) $G_1 - \{a,b\} | c$

Figure 3.1: Example graph $G_1$ and its 3-trees

Although output sensitive, this simple method is not optimal. In fact, a more careful analysis shows that it takes $O((s+1)(m+n))$ time. One problem is that the adjacency lists of $G$ can be of length $O(n)$ each, but we cannot pay such a cost in each recursive call. Also, we need a certificate that should be easily maintained through the recursive calls to guarantee a priori that there will be at least one $k$-subtree generated. By exploiting more refined structural properties of the recursion tree, we present our algorithmic ideas until an optimal output-sensitive listing is obtained, i.e. $O(sk)$ time. Our presentation follows an incremental approach to introduce each idea, so as to evaluate its impact in the complexity of the corresponding algorithms.

3.1 Preliminaries

Given a simple (without self-loops or parallel edges), undirected and connected graph $G = (V,E)$, with $n = |V|$ and $m = |E|$, and an integer $k \in [2,n]$, a $k$-subtree $T$ is an acyclic connected subgraph of $G$ with $k$ vertices. We denote the total number of $k$-subtrees in $G$ by $s$, where $sk \geq m \geq n - 1$ since $G$ is connected. Let us formulate the problem of listing $k$-subtrees.

**Problem 3.1.** Given an input graph $G$ and an integer $k$, list all the $k$-subtrees of $G$.

We say that an algorithm that solves Problem 3.1 is optimal if it takes $O(sk)$ time, since the latter is proportional to the time taken to explicitly list the output,
3.2. BASIC APPROACH: RECURSION TREE

We adopt the standard representation of graphs using adjacency lists \( \text{adj}(v) \) for each vertex \( v \in V \). We maintain a counter for each \( v \in V \), denoted by \( |\text{adj}(v)| \), with the number of edges in the adjacency list of \( v \). Additionally, as the graph is undirected, \((u, v)\) and \((v, u)\) are equivalent.

Let \( X \subseteq E \) be a connected edge set. We denote by \( V[X] = \{u \mid (u, v) \in X\} \) the set of its endpoints, and its ordered vertex list \( \hat{V}(X) \) recursively as follows: \( \hat{V}(\{(\cdot, u_0)\}) = \langle u_0 \rangle \) and \( \hat{V}(X + (u, v)) = \hat{V}(X) + \langle v \rangle \) where \( u \in V[X], \ v \notin V[X] \), and \( + \) denotes list concatenation. We also use the shorthand \( E[X] \equiv \{(u, v) \in E \mid u, v \in V[X]\} \) for the induced edges. In general, \( G[X] = (V[X], E[X]) \) denotes the subgraph of \( G \) induced by \( X \), which is equivalently defined as the subgraph of \( G \) induced by the vertices in \( V[X] \).

The cutset of \( X \) is the set of edges \( C(X) \subseteq E \) such that \((u, v) \in C(X)\) if and only if \( u \in V[X] \) and \( v \in V - V[X] \). Note that when \( V[X] = V \), the cutset is empty. Similarly, the ordered cutlist \( \hat{C}(X) \) contains the edges in \( C(X) \) ordered by the rank of their endpoints in \( \hat{V}(X) \). If two edges have the same endpoint vertex \( v \in \hat{V}(S) \), we use the order in which they appear in \( \text{adj}(v) \) to break the tie.

Throughout the chapter we represent an unordered \( k' \)-subtree \( T = \langle e_1, e_2, \ldots, e_{k'} \rangle \) with \( k' \leq k \) as an ordered, connected and acyclic list of \( k' \) edges, where we use a dummy edge \( e_i = (\cdot, v_i) \) having a vertex \( v_i \) of \( T \) as endpoint. The order is the one by which we discover the edges \( e_1, e_2, \ldots, e_{k'} \). Nevertheless, we do not generate two different orderings for the same \( T \).

3.2 Basic approach: recursion tree

We begin by presenting a simple algorithm that solves Problem 3.1 in \( O(sk^3) \) time, while using \( O(mk) \) space. Note that the algorithm is not optimal yet: we will show in Sections 3.3-3.4 how to improve it to obtain an optimal solution with \( O(m) \) space and delay \( t(k) = k^2 \).

3.2.1 Top level

We use the standard idea of fixing an ordering of the vertices in \( V = \langle v_1, v_2, \ldots, v_n \rangle \). For each \( v_i \in V \), we list the \( k \)-subtrees that include \( v_i \) and do not include any previous vertex \( v_j \in V \) \((j < i)\). After reporting the corresponding \( k \)-subtrees, we remove \( v_i \) and its incident edges from our graph \( G \). We then repeat the process, as summarized in Algorithm I. Here, \( S \) denotes a \( k' \)-subtree with \( k' \leq k \), and we use the dummy edge \((\cdot, v_i)\) as a start-up point, so that the ordered vertex list is \( \hat{V}(S) = \langle v_i \rangle \). Then, we find a \( k \)-subtree by performing a DFS starting from \( v_i \): when we meet the \( k \)th vertex, we are sure that there exists at least one \( k \)-subtree for
v_i and execute the binary partition method with ListTrees_k; otherwise, if there is no such k-subtree, we can skip v_i safely. We exploit some properties on the recursion tree and an efficient implementation of the following operations on G:

- **del(u)** deletes a vertex u ∈ V and all its incident edges.
- **del(e)** deletes an edge e = (u, v) ∈ E. The inverse operation is denoted by undel(e). Note that |adj(v)| and |adj(u)| are updated.
- **choose(S)**, for a k'-subtree S with k' ≤ k, returns an edge e ∈ C(S): e− the vertex in e that belongs to V[S] and by e+ the one s.t. e+ ∈ V − V[S].
- **dfs_k(S)** returns the list of the tree edges obtained by a truncated DFS, where conceptually S is treated as a single (collapsed vertex) source whose adjacency list is the cutset C(S). The DFS is truncated when it finds k tree edges (or less if there are not so many). The resulting list is a k-subtree (or smaller) that includes all the edges in S. Its purpose is to check if there exists a connected component of size at least k that contains S.

**Lemma 3.1.** Given a graph G and a k'-subtree S, we can implement the following operations: del(u) for a vertex u in time proportional to u’s degree; del(e) and undel(e) for an edge e in O(1) time; choose(S) and dfs_k(S) in O(k^2) time.

**Proof.** We represent G using standard adjacency lists where, for each edge (u, v), we keep references to its occurrence in the adjacency list of u and v, respectively. We also update |adj(u)| and |adj(v)| in constant time. This immediately gives the complexity of the operations del(u), del(e), and undel(e).

As for choose(S), recall that S is a k'-subtree with k' < k. Recall that there are at most (k')^2 = O(k^2) edges in between the vertices in V[S]. Hence, it takes O(k^2) to discover an edge that belongs to the cutset C(S): it is the first one found in the adjacency lists of V[S] having an endpoint in V − V[S]. This can be done in O(k^2) time using a standard trick: start from a binary array B of n elements set to 0. For each x ∈ V[S], set B[x] := 1. At this point, scan the adjacency lists of the vertices in V[S] and find a vertex y within them with B[y] = 0. After that, clean it up setting B[x] := 0 for each x ∈ V[S].

Consider now dfs_k(S). Starting from the k'-subtree S, we first need to generate the list L of k − k' vertices in V − V[S] (or less if there are not so many) from the cutset C(S). This is a simple variation of the method above, and it takes O(k^2) time. At this point, we start a regular, truncated DFS-tree from a source (conceptually representing the collapsed S) whose adjacency list is L. During the DFS traversal, when we explore an edge (v, u) from the current vertex v, either u has already been visited or u has not yet been visited. Recall that we stop when new k − k' distinct vertices are visited. Since there are at most (k')^2 = O(k^2) edges in between k vertices, we traverse O(k^2) edges before obtaining the truncated DFS-tree of size k.  

□
3.2. BASIC APPROACH: RECURSION TREE

**Algorithm 1** ListAllTrees\((G = (V, E), k)\)
1: for \(i = 1, 2, \ldots, n - 1\) do
2: \(S := (\cdot, v_i)\)
3: if \(|\text{dfs}_k(S)| = k\) then
4: ListTrees\(_{v_i}(S)\)
5: end if
6: del\(_{v_i}(S)\)
7: end for

**Algorithm 2** ListTrees\(_{v_i}(S)\)
1: if \(|S| = k\) then
2: output\((S)\)
3: return
4: end if
5: \(e := \text{choose}(S)\)
6: ListTrees\(_{v_i}(S + \langle e \rangle)\)
7: del\(_e(S)\)
8: if \(|\text{dfs}_k(S)| = k\) then
9: ListTrees\(_{v_i}(S)\)
10: end if
11: undel\(_e(S)\)

### 3.2.2 Recursion tree and analysis

The recursive binary partition method in Algorithm 2 is quite simple, and takes a \(k'\)-subtree \(S\) with \(k' \leq k\) as input. The purpose is that of listing all \(k\)-subtrees that include all the edges in \(S\) (excluding those with endpoints \(v_1, v_2, \ldots, v_{i-1}\)). The precondition is that we recursively explore \(S\) if and only if there is at least a \(k\)-subtree to be listed. The corresponding recursion tree has some interesting properties that we exploit during the analysis of its complexity. The root of this binary tree is associated with \(S = (\cdot, v_i)\). Let \(S\) be the \(k'\)-subtree associated with a node in the recursion tree. Then, left branching occurs by taking an edge \(e \in C(S)\) using \text{choose}, so that the left child is \(S + \langle e \rangle\). Right branching occurs when \(e\) is deleted using \text{del}, and the right child is still \(S\) but on the reduced graph \(G := (V, E - \{e\})\). Returning from recursion, restore \(G\) using \text{undel}(e). Note that we do not generate different permutations of the same \(k'\)-subtree’s edges as we either take an edge \(e\) as part of \(S\) or remove it from the graph by the binary partition method.

**Lemma 3.2.** Algorithm 2 lists each \(k\)-subtree containing vertex \(v_i\) and no vertex \(v_j\) with \(j < i\), once and only once.

**Proof.** ListTrees\(_{v_i}\) outputs all the wanted \(k\)-subtrees according to a simple rule: first list all the \(k\)-subtrees that include edge \(e\) and then those not including \(e\): an
edge \( e \) must exist because of the precondition, and we can choose any edge \( e \) incident to the partial solution \( S \). As \textbf{choose} returns an edge \( e \) from the cutset \( C(S) \), this edge is incident in \( S \) and does not introduce a cycle. Note that if \( \text{dfs}_k(S) \) has size \( k \), then there is a connected component of size \( k \). Hence, there must be a \( k \)-subtree to be listed, as the spanning tree of the component is a valid \( k \)-subtree. Additionally, if there is a \( k \)-subtree to be listed, a connected component of size \( k \) exists. As a result, we conceptually partition all the \( k \)-subtrees in two disjoint sets: the \( k \)-subtrees that include \( S + \langle e \rangle \), and the \( k \)-subtrees that include \( S \) and do not include \( e \). We stop when \(|S| = k \) and we start the partial solution \( S \) with a dummy edge that connects to \( v_i \), ensuring that all trees of size \( k \) connected to \( v_i \) are listed. Since all the vertices \( v_j \) with \( j < i \) are removed from \( G \), \( k \)-subtrees that include \( v_j \) are not listed twice. Therefore, each tree is listed at most one time.

A closer look at the recursion tree (e.g. Figure 3.2), reveals that it is \( k \)-left-bounded: namely, each root-to-leaf path has exactly \( k - 1 \) left branches. Since there is a one-to-one correspondence between the leaves and the \( k \)-subtrees, we are guaranteed that leftward branching occurs less than \( k \) times to output a \( k \)-subtree.

What if we consider rightward branching? Note that the height of the tree is less than \( m \), so we might have to branch rightward \( O(m) \) times in the worst case. Fortunately, we can prove in Lemma 3.3 that for each internal node \( S \) of the recursion tree that has a right child, \( S \) has always its left child (which leads to one \( k \)-subtree). This is subtle but very useful in our analysis in the rest of the chapter.

\textbf{Lemma 3.3.} At each node \( S \) of the recursion tree, if there exists a \( k \)-subtree (descending from \( S \)'s right child) that does not include edge \( e \), then there is a \( k \)-subtree (descending from \( S \)'s left child) that includes \( e \).

\textbf{Proof.} Consider a \( k \)-subtree \( T \) that does not include \( e = (u, v) \), which was opted out at a certain node \( S \) during the recursion. Either \( e \) is only incident to one vertex in \( T \), so there is at least one \( k \)-subtree \( T' \) that includes \( e \) and does not include an edge
3.2. BASIC APPROACH: RECURSION TREE

of \( T \). Or, when \( e = (u, v) \) is incident to two different nodes \( u, v \in V[T] \), there is a valid \( k \)-subtree \( T' \) that includes \( e \) and does not include one edge on the path that connects \( u \) and \( v \) using edges from \( T \). Note that both \( T \) and \( T' \) are “rooted” at \( v \) and are found in two descending leaves from \( S \).

Note that the symmetric situation for Lemma 3.3 does not necessarily hold. We can find nodes having just the left child: for these nodes, the chosen edge cannot be removed since this gives rise to a connected component of size smaller than \( k \). We can now state how many nodes there are in the recursion tree.

**Corollary 3.2.** Let \( s_i \) be the number of \( k \)-subtrees reported by \( \text{ListTrees}_{v_i} \). Then, its recursion tree is binary and contains \( s_i \) leaves and at most \( s_i k \) internal nodes. Among the internal nodes, there are \( s_i - 1 \) of them having two children.

**Proof.** The number \( s_i \) of leaves derives from the one-to-one correspondence with the \( k \)-subtrees found by \( \text{ListTrees}_{v_i} \). To give an upper bound on the number of internal nodes, consider a generic node \( S \) and apply Lemma 3.3 to it. If \( S \) has a single child, it is a left child that leads to one or more \( k \)-subtrees in its descending leaves. So, we can charge one token (corresponding to \( S \)) to the leftmost of these leaves. Hence, the total number of tokens over all the \( s_i \) leaves is at most \( s_i (k - 1) \) since the recursion tree is \( k \)-left-bounded. The other option is that \( S \) has two children: in this case, the number of these kind of nodes cannot exceed the number \( s_i \) of leaves. Summing up, we have a total of \( s_i k \) internal nodes in the recursion tree. Consider the compacted recursion tree, where each maximal path of unary nodes (i.e. internal nodes having one child) is replaced by a single edge. We obtain a binary tree with \( s_i \) leaves and all the other nodes having two children: we fall within a classical situation, for which the number of nodes with two children is one less than the number of leaves, hence, \( s_i - 1 \).

**Lemma 3.4.** Algorithm 2 takes \( O(s_i k^3) \) time and \( O(mk) \) space, where \( s_i \) is the number of \( k \)-subtrees reported by \( \text{ListTrees}_{v_i} \).

**Proof.** Each call to \( \text{ListTrees}_{v_i} \) invokes operations \( \text{del}, \text{undel}, \text{choose}, \) and \( \text{dfs}_k \) once. By Lemma 3.1 the execution time of the call is therefore \( O(k^2) \). Since there are \( O(s_i k) \) calls by Corollary 3.2 the total running time of Algorithm 2 is \( O(s_i k^3) \). The total space of \( O(mk) \) derives from the fact that the height of the recursion tree is at most \( m \). On each node in the recursion path we keep a copy of \( S \) and \( D \), taking \( O(k) \) space. As we modify and restore the graph incrementally, this totals \( O(mk) \) space.

**Theorem 3.3.** Algorithm 2 can solve Problem 3.1 in \( O(nk^2 + sk^3) = O(sk^3) \) time and \( O(mk) \) space.

**Proof.** The correctness of Algorithm 1 easily derives from Lemma 3.2 so it outputs each \( k \)-subtree once and only once. Its cost is upper bounded by the sum (over
A way to improve the running time of \( \text{ListTrees}_v \) is indirectly suggested by Corollary 3.2. Since there are \( O(s_i) \) binary nodes and \( O(s_i k) \) unary nodes in the recursion tree, we can pay \( O(k^2) \) time for binary nodes and \( O(1) \) for unary nodes (i.e. reduce the cost of \( \text{choose} \) and \( \text{dfs}_k \) to \( O(1) \) time when we are in a unary node). This way, the total running time is \( O(s k^2) \).

The idea is to maintain a certificate that can tell us if we are in a unary node in \( O(1) \) time and that can be updated in \( O(1) \) time in such a case, or can be completely rebuilt in \( O(k^2) \) time otherwise (i.e. for binary nodes). This will guarantee a total cost of \( O(s_i k^2) \) time for \( \text{ListTrees}_v \), and lay out the path to the wanted optimal output-sensitive solution of Section 3.4.

### 3.3 Improved approach: certificates

A way to improve the running time of \( \text{ListTrees}_v \) to \( O(s_i k^2) \) is indirectly suggested by Corollary 3.2. Since there are \( O(s_i) \) binary nodes and \( O(s_i k) \) unary nodes in the recursion tree, we can pay \( O(k^2) \) time for binary nodes and \( O(1) \) for unary nodes (i.e. reduce the cost of \( \text{choose} \) and \( \text{dfs}_k \) to \( O(1) \) time when we are in a unary node). This way, the total running time is \( O(s k^2) \).

The idea is to maintain a certificate that can tell us if we are in a unary node in \( O(1) \) time and that can be updated in \( O(1) \) time in such a case, or can be completely rebuilt in \( O(k^2) \) time otherwise (i.e. for binary nodes). This will guarantee a total cost of \( O(s_i k^2) \) time for \( \text{ListTrees}_v \), and lay out the path to the wanted optimal output-sensitive solution of Section 3.4.

### 3.3.1 Introducing certificates

We impose an “unequivocal behavior” to \( \text{dfs}_k(S) \), obtaining a variation denoted \( \text{mdfs}_k(S) \) and called \textit{multi-source truncated DFS}. During its execution, \( \text{mdfs}_k \) takes the order of the edges in \( S \) into account (whereas an order is not strictly necessary in \( \text{dfs}_k \)). Specifically, given a \( k \)-subtree \( S = \{e_1, e_2, \ldots, e_{m'}\} \), the returned \( k \)-subtree \( D = \text{mdfs}_k(S) \) contains \( S \), which is conceptually treated as a collapsed vertex: the main difference is that \( S \)’s “adjacency list” is now the \textit{ordered cutlist} \( \hat{C}(S) \), rather than \( C(S) \) employed for \( \text{dfs}_k \).

Equivalently, since \( \hat{C}(S) \) is induced from \( C(S) \) by using the ordering in \( \hat{V}(S) \), we can see \( \text{mdfs}_k(S) \) as the execution of multiple standard DFSes from the vertices in \( \hat{V}(S) \), in that order. Also, all the vertices in \( V[S] \) are conceptually marked as visited at the beginning of \( \text{mdfs}_k \), so \( u_j \) is never part of the DFS tree starting from \( u_i \) for any two distinct \( u_i, u_j \in V[S] \). Hence the adopted terminology of multi-source. Clearly, \( \text{mdfs}_k(S) \) is a feasible solution to \( \text{dfs}_k(S) \) while the vice versa is not true.

We use the notation \( S \subseteq D \) to indicate that \( D = \text{mdfs}_k(S) \), and so \( D \) is a \textit{certificate} for \( S \); it guarantees that node \( S \) in the recursion tree has at least one descending leaf whose corresponding \( k \)-subtree has not been listed so far. Since the
behavior of $\text{mdfs}_k$ is non-ambiguous, relation $\sqsubseteq$ is well defined. We preserve the following invariant on $\text{ListTrees}_{v_i}$, which now has two arguments.

**Invariant 1.** For each call to $\text{ListTrees}_{v_i}(S, D)$, we have $S \sqsubseteq D$.

Before showing how to keep the invariant, we detail how to represent the certificate $D$ in a way that it can be efficiently updated. We maintain it as a partition $D = S \cup L \cup F$, where $S$ is the given list of edges, whose endpoints are kept in order as $\hat{V}(S) = \langle u_1, u_2, \ldots, u_k \rangle$. Moreover, $L = D \cap C(S)$ are the tree edges of $D$ in the cutset $C(S)$, and $F$ is the forest storing the edges of $D$ whose both endpoints are in $V[D] - V[S]$.

(i) We store the $k'$-subtree $S$ as a sorted doubly-linked list of $k'$ edges $\langle e_1, e_2, \ldots, e_{k'} \rangle$, where $e_1 := (\cdot, v_i)$. We also keep the sorted doubly-linked list of vertices $\hat{V}(S) = \langle u_1, u_2, \ldots, u_{k'} \rangle$ associated with $S$, where $u_1 := v_i$. For $1 \leq j \leq k'$, we keep the number of tree edges in the cutset that are incident to $u_j$, namely $\eta[u_j] = |\{ (u_j, x) \in L \}|$.

(ii) We keep $L = D \cap C(S)$ as an ordered doubly-linked list of edges in $\hat{C}(S)$’s order; it can be easily obtained by maintaining the parent edge connecting a root in $F$ to its parent in $\hat{V}(S)$.

(iii) We store the forest $F$ as a sorted doubly-linked list of the roots of the trees in $F$. The order of this list is that induced by $\hat{C}(S)$: a root $r$ precedes a root $t$ if the (unique) edge in $L$ incident to $r$ appears before the (unique) edge of $L$ incident to $t$. For each node $x$ of a tree $T \in F$, we also keep its number $\text{deg}(x)$ of children in $T$, and its predecessor and successor sibling in $T$.

(iv) We maintain a flag $\text{is\_unary}$ that is true if and only if $|\text{adj}(u_i)| = \eta[u_i] + \sigma(u_i)$ for all $1 \leq i \leq k'$, where $\sigma(u_i) = |\{ (u_i, u_j) \in E \mid i \neq j \}|$ is the number of internal edges, namely, having both endpoints in $V[S]$.

Throughout the chapter, we identify $D$ with both (1) the set of $k$ edges forming it as a $k$-subtree and (2) its representation above as a certificate. We also support the following operations on $D$, under the requirement that $\text{is\_unary}$ is true (i.e. all the edges in the cutset $C(S)$ are tree edges), otherwise they are undefined:

- $\text{treecut}(D)$ returns the last edge in $L$.
- $\text{promote}(r, D)$, where root $r$ is the last in the doubly-linked list for $F$: remove $r$ from $F$ and replace $r$ with its children $r_1, r_2, \ldots, r_c$ (if any) in that list, so they become the new roots (and so $L$ is updated).

**Lemma 3.5.** The representation of certificate $D = S \cup L \cup F$ requires $O(|D|) = O(k)$ memory words, and $\text{mdfs}_k(S)$ can build $D$ in $O(k^2)$ time. Moreover, each of the operations $\text{treecut}$ and $\text{promote}$ can be supported in $O(1)$ time.
Proof. Recall that $D = S \cup L \cup F$ and that $|S| + |L| + |F| = k$ when they are seen as sets of edges. Hence, $D$ requires $O(k)$ memory words, since the representation of $S$ in the certificate takes $O(|S|)$ space, $L$ takes $O(|L|)$ space, $F$ takes $O(|F|)$ space, and is\_unary takes $O(1)$ space.

Building the representation of $D$ takes $O(k^2)$ using mdfs\_k. Note that the algorithm behind mdfs\_k is very similar to dfs\_k (see the proof of Lemma 3.2.1) except that the order in which the adjacency lists for the vertices in $V[S]$ are explored is that given by $\hat{V}(S)$. After that the edges in $D$ are found in $O(k^2)$ time, it takes no more than $O(k^2)$ time to build the lists in points [i]–[iii] and check the condition in point [iv] of Section 3.3.1

Operation treecut is simply implemented in constant time by returning the last edge in the list for $L$ (point [ii] of Section 3.3.1).

As for promote($r, D$), the (unique) incident edge $(x, r)$ that belongs to $L$ is removed from $L$ and added to $S$ (updating the information in point [i] of Section 3.3.1), and edges $(r, r_1), \ldots, (r, r_c)$ are appended at the end of the list for $L$ to preserve the cutlist order (updating the information in [ii]). This is easily done using the sibling list of $r$’s children: only a constant number of elements need to be modified. Since we know that is\_unary is true before executing promote, we just need to check if appending $r$ to $\hat{V}(S)$ adds non-tree edges to the cutlist $\hat{C}(S)$. Recalling that there are $\deg(r) + 1$ tree edges incident to $r$, this is only the case when $|\text{adj}(r)| > \deg(r) + 1$, which can be easily checked in $O(1)$ time: if so, we set is\_unary to false, otherwise we leave is\_unary true. Finally, we correctly set $\eta[r] := \deg(r)$, and decrease $\eta[x]$ by one, since is\_unary was true before executing promote.

3.3.2 Maintaining the invariant

We now define choose in a more refined way to facilitate the task of maintaining the invariant $S \subseteq D$ introduced in Section 3.3.1. As an intuition, choose selects an edge $e = (e^-, e^+)$ from the cutlist $\hat{C}(S)$ that interferes as least as possible with the certificate $D$. Recalling that $e^- \in V[S]$ and $e^+ \in V - V[S]$ by definition of cutlist, we consider the following case analysis:

(a) [external edge] Check if there exists an edge $e \in \hat{C}(S)$ such that $e \notin D$ and $e^+ \notin V[D]$. If so, return $e$, shown as a saw in Figure 3.3(a).

(b) [back edge] Otherwise, check if there exists an edge $e \in \hat{C}(S)$ such that $e \notin D$ and $e^+ \in V[D]$. If so, return $e$, shown dashed in Figure 3.3(b).

(c) [tree edge] As a last resort, every $e \in \hat{C}(S)$ must be also $e \in D$ (i.e. all edges in the cutlist are tree edges). Return $e := \text{treecut}(D)$, the last edge from $\hat{C}(S)$, shown as a coil in Fig. 3.3(c).

Lemma 3.6. For a given $k'$-subtree $S$, consider its corresponding node in the recursion tree. Then, this node is binary when choose returns an external or back edge (cases [a]–[b]) and is unary when choose returns a tree edge (case [c]).
3.3. IMPROVED APPROACH: CERTIFICATES

Figure 3.3: Choosing edge \( e \in C(S) \). The certificate \( D \) is shadowed.

Algorithm 3 ListAllTrees( \( G = (V,E), k \) )

1: for \( v_i \in V \) do
2: \( S := \langle \cdot, v_i \rangle \)
3: \( D := \text{mdfs}_k(S) \)
4: if \( |D| < k \) then
5: for \( u \in V[D] \) do
6: \( \text{del}(u) \)
7: end for
8: else
9: ListTrees\(_{v_i}(S, D)\)
10: \( \text{del}(v_i) \)
11: end if
12: end for

Proof. Consider a node \( S \) in the recursion tree and the corresponding certificate \( D \). For a given edge \( e \) returned by \text{choose}(S, D)\) note that: if \( e \) is an external or back edge (cases (a)–(b)), \( e \) does not belong to the \( k \)-subtree \( D \) and therefore there exists a \( k \)-subtree that does not include \( e \). By Lemma 3.3, a \( k \)-subtree that includes \( e \) must exist and hence the node is binary. We are left with case (c) where \( e \) is an edge of \( D \) that belongs to the cutlist \( \hat{C}(S) \). Recall that, by the way \text{choose} proceeds, all edges in the cutlist \( \hat{C}(S) \) belong to \( D \) (see Figure 3.3(c)). There are no further \( k \)-subtrees that do not include edge \( e \) as \( e \) is the last edge from \( \hat{C}(S) \) in the order of the truncated DFS tree, and so the node in the recursion tree is unary. The existence of a \( k \)-subtree that does not include \( e \) would imply that \( D \) is not the valid \( \text{mdfs}_k(S) \): at least \( k \) vertices would be reachable using the previous branches of \( \hat{C}(S) \) which is a contradiction with the fact that we traverse vertices in the DFS order of \( \hat{V}(S) \). \( \square \)
We now present the new listing approach in Algorithm 3. If the connected component of vertex $v_i$ in the residual graph is smaller than $k$, we delete its vertices since they cannot provide $k$-subtrees, and so we skip them in this way. Otherwise, we launch the new version of ListTrees, shown in Algorithm 4. In comparison with the previous version (Algorithm 2), we produce the new certificate $D'$ from the current $D$ in $O(1)$ time when we are in a unary node. On the other hand, we completely rebuild the certificate twice when we are in a binary node (since either child could be unary at the next recursion level).

Lemma 3.7. Algorithm 4 correctly maintains the invariant $S \subseteq D$.

Proof. The base case is when $|S| = k$, as before. Hence, we discuss the recursion, where we suppose that $S$ is a $k'$-subtree with $k' < k$. Let $e = (e^-, e^+)$ denote the edge returned by $\text{choose}(S, D)$.

First: Consider the situation in which we want to output all the $k$-subtrees that contain $S' = S + \langle e \rangle$. Now, from certificate $D = S \cup L \cup F$ we obtain a new $D' = S' \cup L' \cup F'$ according to the three cases behind $\text{choose}$, for which we have to prove that $S' \subseteq D'$.

- [a] [external edge] We simply recompute $D' := \text{mdfs}_k(S')$. So $S' \subseteq D'$ by definition of relation $\subseteq$.
- [b] [back edge] Same as in the case above, we recompute $D' := \text{mdfs}_k(S')$ and therefore $S' \subseteq D'$.
- [c] [tree edge] In this case, the set of edges of the certificate does not change ($D' = D$ seen as sets), but the internal representation described in Section 3.3.1
3.3. IMPROVED APPROACH: CERTIFICATES

changes partially since \( S' = S + \langle e \rangle \). The flag is\_unary is true, and so treecut and promote can be invoked. The former is done by choose, which correctly returns \( e = (e^-, e^+) \) as the last tree edge in the cutlist \( \hat{C}(S) \). The latter is done to promote the children \( r_1, r_2, \ldots, r_c \) of \( e^+ \).

To show that \( S' \sqsubseteq D' \) for this case, we need to prove that the resulting certificate \( D' \) is the same as the one returned by an explicit call to mdfs\(_k\)(\( S' \)), which we clearly want to avoid calling. Let \( D_0 = S_0 \cup L_0 \cup F_0 \) be the output of the call to mdfs\(_k\)(\( S' \)), and let \( D' = S' \cup L' \cup F' \) be what we obtain in Algorithm 4.

First of all, note that \( S' = S_0 = S + \langle e \rangle \) by definition of mdfs\(_k\). This means that the sorted lists for \( S' \) and \( V(S') \) are “equal” to those for \( S_0 \) and \( V(S_0) \) (elements are the same and in the same order). Hence, \( S' = S_0 = \langle e_1, e_2, \ldots, e_{k'}, e \rangle \) and \( V(S') = V(S_0) = \langle u_1, u_2, \ldots, u_{k'}, e \rangle \).

Consequently, the cutsets \( C(S') = C(S_0) \): when considering the corresponding cutlists \( \hat{C}(S') \) and \( \hat{C}(S_0) \), recall that mdfs\(_k\) performs a multi-source truncated DFS from the vertices \( u_1, u_2, \ldots, u_{k'}, e^+ \) in this order (where all of them are initially marked as already visited). When mdfs\(_k\) starts from \( e^- = u_j \) (for some \( 1 \leq j \leq k' \)), it does not explore \( e^+ \) through edge \( e \). Moreover, \( r_1, r_2, \ldots, r_c \) are not explored as well, since otherwise there would be back edges and is\_unary would be false. Since \( e^+ \) is the last in \( S' \), when mdfs\(_k\) starts from \( e^+ \), observe that \( e^- \) has been totally explored, and \( r_1, r_2, \ldots, r_c \) are discovered now from \( e^+ \). Since \( e \) is the last tree edge in the cutlist \( \hat{C}(S) \), we have that the ordering in the new cutlists \( \hat{C}(S') \) and \( \hat{C}(S_0) \) must be the same.

Consider now \( L' \) and \( L_0 \). We show that \( L' = L_0 \) using the fact that \( \hat{C}(S') = \hat{C}(S_0) \). Operation promote\((e^+, D)\) removes \( e \) from \( L \) and adds tree edges \( (e^+, r_i) \) for its children \( r_1, r_2, \ldots, r_c \) to form \( L' \). Note these edges are added in the same order as they were discovered by mdfs\(_k\)(\( S \)) and mdfs\(_k\)(\( S' \)) since is\_unary is true and \( \hat{C}(S') = \hat{C}(S_0) \). Since \( L_0 \) does not contain \( e \), we have that \( L' = L_0 \).

It remains to show that \( F' = F_0 \). This is easy since \( \hat{C}(S') = \hat{C}(S_0) \): the subtree at \( r_i \) is totally explored before that at \( r_j \) for \( i \leq j \). Hence, when \( r_1, r_2, \ldots, r_c \) are promoted as roots in \( F' \), their corresponding subtrees do not change. Also, their ordering in the sublist for \( F' \) and \( F_0 \) is the same because \( \hat{C}(S') = \hat{C}(S_0) \).

Finally, since there are no back edges, the update of \( \eta \) only involves \( \eta[e^-] \) and \( \eta[e^+] \) as discussed in the implementation of promote. For the same reason, the only case in which is\_unary can become false is when \( |\text{adj}(e^+)| > \deg(e^+) + 1 \). This completes the proof that \( S' \sqsubseteq D' \) for case [c].

Second: Consider the situation in which we want to list all the \( k \)-subtrees that contain \( S \) but do not contain \( e \). This is equivalent to list all the \( k \)-subtrees that contain \( S \) in \( G - \{e\} \). Hence, we remove \( e \) from \( G \) and recomputed the certificate from scratch before each of the two recursive calls. Consider the three cases behind choose. Cases [a] and [b] are trivial, since we recompute \( D'' := \text{mdfs}_k(S) \) and so \( S \sqsubseteq D'' \). Case [c] cannot arise by Lemma 3.6.
3.3.3 Analysis

We implement choose\((S, D)\) so that it can now exploit the information in \(D\). At each node \(S\) of the recursion tree, when it selects an edge \(e\) that belongs to the cutset \(C(S)\), it first considers the edges in \(C(S)\) that are external or back (cases \([a]\), \([b]\)) before the edges in \(D\) (case \([c]\)).

**Lemma 3.8.** There is an implementation of choose in \(O(1)\) for unary nodes in the recursion tree and \(O(k^2)\) for binary nodes.

**Proof.** Given \(D\), we can check if the current node \(S\) in the recursion tree is unary by checking the flag is_unary. If this is the case we simply return the edge indicated by treecut\((D)\) in \(O(1)\) time. Otherwise, the node \(S\) is binary, and so there exists at least an external edge or a back edge. We visit the first \(2k\) edges in each \(\text{adj}(u)\) for every \(u \in S\). Note that less than \(k\) edges can connect \(u\) to vertices in \(V[S]\) and less than \(k\) edges can connect \(u\) to vertices in \(V[D]\): if an external edge exists, we can find it in \(O(k^2)\) time. Otherwise, no external edge exists, so there must be a back edge to be returned since the node is binary. We visit the first \(k\) edges in each \(\text{adj}(u)\) for every \(u \in S\), and surely find one back edge in \(O(k^2)\).

**Lemma 3.9.** Algorithm 4 takes \(O(s_ik^2)\) time and \(O(mk)\) space, where \(s_i\) is the number of \(k\)-subtrees reported by ListTrees\(_{vi}\).

**Proof.** We report the breakdown of the costs for a call to ListTrees\(_{vi}\) according the cases, using Lemmas 3.5 and 3.8:

- \([a]\) External edge: \(O(k^2)\) for choose and mdfs\(_k\), \(O(1)\) for del, undel.
- \([b]\) Back edge: \(O(k^2)\) for choose and mdfs\(_k\), and \(O(1)\) for del and undel.
- \([c]\) Tree edge: \(O(1)\) for choose and promote.

Hence, binary nodes take \(O(k^2)\) time and unary nodes take \(O(1)\) time. By Corollary 3.2 there are \(O(s_i)\) binary nodes and \(O(s_k)\) unary nodes, and so Algorithm 4 takes \(O(s_ik^2)\) time. The space analysis is left unchanged, namely, \(O(mk)\) space.

**Theorem 3.4.** Algorithm 3 solves Problem 3.1 in \(O(sk^2)\) time and \(O(mk)\) space.

**Proof.** The vertices belonging to the connected components of size less than \(k\) in the residual graph, now contribute with \(O(m)\) total time rather than \(O(nk^2)\). The rest of the complexity follows from Lemma 3.9.

3.4 Optimal approach: amortization

In this section, we discuss how to adapt Algorithm 4 so that a more careful analysis can show that it takes \(O(sk)\) time to list the \(k\)-subtrees. Considering ListTrees\(_{vi}\),
observe that each of the \(O(s_k)\) unary nodes requires a cost of \(O(1)\) time and therefore they are not much of problem. On the contrary, each of the \(O(s_i)\) binary nodes takes \(O(k^2)\) time: our goal is to improve this case.

Consider the operations on a binary node \(S\) of the recursion tree that take \(O(k^2)\) time, namely: (I) \(e := \text{choose}(S, D)\); (II) \(D' := \text{mdfs}_k(S')\), where \(S' \equiv S + \langle e \rangle\); and (III) \(D'' := \text{mdfs}_k(S)\) in \(G - \{e\}\). In all these operations, while scanning the adjacency lists of vertices in \(V[S]\), we visit some edges \(e' = (u, v)\), named internal, such that \(e' \not\in S\) with \(u, v \in V[S]\). These internal edges of \(V[S]\) can be visited even if they were previously visited on an ancestor node to \(S\). In Section 3.4.1 we show how to amortize the cost induced by the internal edges. In Section 3.4.2 we show how to amortize the cost induced by the remaining edges and obtain a delay of \(t(k) = k^2\) in our optimal output-sensitive algorithm.

3.4.1 Internal edges

To avoid visiting the internal edges of \(V[S]\) several times throughout the recursion tree, we remove these edges from the graph \(G\) on the fly, and introduce a global data structure, which we call parking lists, to store them temporarily. Indeed, out of the possible \(O(n)\) incident edges in vertex \(u \in V[S]\), less than \(k\) are internal: it is simply too costly removing these internal edges by a complete scan of \(\text{adj}(u)\). Therefore we remove them as they appear while executing \text{choose} and \text{mdfs}_k operations.

Formally, we define parking lists as a global array \(P\) of \(n\) pointers to lists of edges, where \(P[u]\) is the list of internal edges discovered for \(u \in V[S]\). When \(u \not\in V[S]\), \(P[u]\) is null. On the implementation level, we introduce a slight modification of the \text{choose} and \text{mdfs}_k algorithms such that, when they meet for the first (and only) time an internal edge \(e' = (u, v)\) with \(u, v \in V[S]\), they perform \(\text{del}(e')\) and add \(e'\) at the end of both parking lists \(P[u]\) and \(P[v]\). We also keep a cross reference to the occurrences of \(e'\) in these two lists.

Additionally, we perform a small modification in algorithm \text{ListTrees}_{vi} by adding a fifth step in Algorithm [2] just before it returns to the caller. Recall that on the recursion node \(S + \langle e \rangle\) with \(e = (e^-, e^+)\), we added the vertex \(e^+\) to \(V[S]\). Therefore, when we return from the call, all the internal edges incident to \(e^+\) are no longer internal edges (and are the only internal edges to change status). On this new fifth step, we scan \(P[e^+]\) and for each edge \(e' = (e^+, x)\) in it, we remove \(e'\) from both \(P[e^+]\) and \(P[x]\) in \(O(1)\) time using the cross reference. Note that when the node is unary there are no internal edges incident to \(e^+\), so \(P[e^+]\) is empty and the total cost is \(O(1)\). When the node is binary, there are at most \(k - 1\) edges in \(P[e^+]\), so the cost is \(O(k)\).

Lemma 3.10. The operations over internal edges done in \text{ListTrees}_{vi} have a total cost of \(O(s_k)\) time.

Proof. For each \(\text{del}(e')\) in \text{choose} or \text{mdfs}_k, there is exactly one matching \(\text{undel}(e')\) in \text{ListTrees}_{vi} done in a binary node. Since there are \(O(s_i)\) binary nodes and
$O(k)$ unde1 calls per binary node, the total contribution of internal edges to the complexity can be overall bounded by $O(s,k)$ time.

3.4.2 Amortization

Let us now focus on the contribution given by the remaining edges, which are not internal for the current $V[S]$. Given the results in Section 3.4.1 for the rest of this section we can assume wlog that there are no internal edges in $V[S]$, namely, $E[S] = S$. We introduce two metrics that help us to parameterize the time complexity of the operations done in binary nodes of the recursion tree.

The first metric we introduce is helpful when analyzing the operation choose. For connected edge sets $S$ and $X$ with $S \subseteq X$, define the cut number $\gamma_X$ as the number of edges in the induced (connected) subgraph $G[X] = (V[X], E[X])$ that are in the cutset $C(S)$ (i.e. tree edges plus back edges): $\gamma_X = |E[X] \cap C(S)|$.

**Lemma 3.11.** For a binary node $S$ with certificate $D$, choose $(S,D)$ takes $O(k + \gamma_D)$ time.

**Proof.** Consider that we are in a binary node $S$ with associated certificate $D$, otherwise just return treecut($D$) in $O(1)$. We examine each adjacency list $\text{adj}(u)$ for $u \in V[S]$ until we find an external edge. If we scan all these lists completely without finding an external edge, we have visited less than $k$ edges that belong to $S$ and $\gamma_D$ edges that belong to $C(S)$ (the tree and back edges of $D$). When we know that no external edges exist, a back edge is found in $O(k)$ time. This totals $O(k + \gamma_D)$ time.

For connected edge sets $S$ and $X$ with $S \subseteq X$, the second metric is the cyclomatic number $\nu_X$ (also known as circuit rank, nullity, or dimension of cycle space) as the smallest number of edges which must be removed from $G[X]$ so that no cycle remains in it: $\nu_X = |E[X]| - |V[X]| + 1$.

Using the cyclomatic number of a certificate $D$ (ignoring the internal edges of $V[S]$), we obtain a lower bound on the number of $k$-subtrees that are output in the leaves descending from a node $S$ in the recursion tree.

**Lemma 3.12.** Considering the cyclomatic number $\nu_D$ and the fact that $|V[D]| = k$, we have that $G[D]$ contains at least $\nu_D$ $k$-subtrees.

**Proof.** As $G[D]$ is connected, it contains at least one spanning tree. Take any spanning tree $T$ of $G[D]$. Note that there are $\nu_D$ edges in $G[D]$ that do not belong to this spanning tree. For each of these $\nu_D$ edges $e_1, \ldots, e_\nu$, one can construct a spanning tree $T_i$ by adding $e_i$ to $T$ and breaking the cycle introduced. These $\nu_D$ spanning trees are all different $k$-subtrees as they include a different edge $e_i$ and have $k$ different edges.

**Lemma 3.13.** For a node $S$ with certificate $D$, computing $D' = \text{mdfs}_k(S)$ takes $O(k + \nu_D')$ time.
3.4. OPTIMAL APPROACH: AMORIZATION

Proof. Consider the certificate $k$-subtree $D'$ returned by $\text{mdfs}_k(S)$. To calculate $D'$, the edges in $D'$ are visited and, in the worst case, all the $\nu_{D'}$ edges in between vertices of $V[D']$ in $G$ are also visited. No other edge $e' = (u, v)$ with $u \in D'$ and $v \not\in D'$ is visited, as $v$ would belong to $D'$ since we visit vertices depth first. \hfill \Box

Recalling that the steps done on a binary node $S$ with certificate $D$ are: (I) $e := \text{choose}(S, D)$; (II) $D' := \text{mdfs}_k(S')$, where $S' := S + \langle e \rangle$; and (III) $D'' := \text{mdfs}_k(S)$ in $G - \{e\}$, they take a total time of $O(k + \gamma_D + \nu_{D'} + \nu_{D''})$. We want to pay $O(k)$ time on the recursion node $S$ and amortize the rest of the cost to some suitable nodes descending from its left child $S'$ (with certificate $D'$). To do this we are to relate $\gamma_D$ with $\nu_{D'}$ and avoid performing step (III) in $G - \{e\}$ by maintaining $D''$ from $D'$. We exploit the property that the cost $O(k + \nu_{D'})$ for a node $S$ in the recursion tree can be amortized using the following lemma:

Lemma 3.14. Let $S'$ be the left child (with certificate $D'$) of a generic node $S$ in the recursion tree. The sum of $O(\nu_{D'})$ work, over all left children $S'$ in the recursion tree is upper bounded by $\sum_{S'} \nu_{D'} = O(s_i k)$.

Proof. By Lemma 3.12, $S'$ has at least $\nu_{D'}$ descending leaves. Charge $O(1)$ to each leaf descending from $S'$ in the recursion tree. Since $S'$ is a left child and we know that the recursion tree is $k$-left-bounded by Lemma 3.3, each of the $s_i$ leaves can be charged at most $k$ times, so $\sum_{S'} \nu_{D'} = O(s_i k)$ for all such $S'$.

We now show how to amortize the $O(k + \gamma_D)$ cost of step (I). Let us define $\text{comb}(S')$ for a left child $S'$ in the recursion tree as its maximal path to the right (its right spine) and the left child of each node in such a path. Then, $|\text{comb}(S')|$ is the number of such left children.

Lemma 3.15. On a node $S$ in the recursion tree, the cost of $\text{choose}(S, D)$ is $O(k + \gamma_D) = O(k + \nu_{D'} + |\text{comb}(S')|)$.

Proof. Consider the set $E'$ of $\gamma_D$ edges in $E[D] \cap C(S)$]. Take $D'$, which is obtained from $S' = S + \langle e \rangle$, and classify the edges in $E'$ accordingly. Given $e' \in E'$, one of three possible situations may arise: either $e'$ becomes a tree edge part of $D'$ (and so it contributes to the term $k$), or $e'$ becomes a back edge in $G[D']$ (and so it contributes to the term $\nu_{D'}$), or $e'$ becomes an external edge for $D'$. In the latter case, $e'$ will be chosen in one of the subsequent recursive calls, specifically one in $\text{comb}(S')$ since $e'$ is still part of $C(S')$ and will surely give rise to another $k$-subtree in a descending leaf of $\text{comb}(S')$.

While the $O(\nu_{D'})$ cost over the leaves of $S'$ can be amortized by Lemma 3.13, we need to show how to amortize the cost of $|\text{comb}(S')|$ using the following:

Lemma 3.16. $\sum_{S'} |\text{comb}(S')| = O(s_i k)$ over all left children $S'$ in the recursion.
Proof. Given a left child $S^*$ in the recursion tree, there is always a unique node
$S'$ that is a left child such that its $\text{comb}(S')$ contains $S^*$. Hence, $\sum_{S'} |\text{comb}(S')|$ is upper bounded by the number of left children in the recursion tree, which is $O(s_ik)$.

At this point we are left with the cost of computing the two $\text{mdfs}_k$’s. Note that the cost of step (II) is $O(k + \nu_{D'})$, and so is already expressed in terms of the cyclomatic number of its left child, $\nu_{D'}$ (so we use Lemma 3.12). The cost of step (III) is $O(k + \nu_{D''})$, expressed with the cyclomatic number of the certificate of its right child. This cost is not as easy to amortize since, when the edge $e$ returned by $\text{choose}$ is a back edge, $D'$ of node $S + (e)$ can change heavily causing $D'$ to have just $S$ in common with $D''$. This shows that $\nu_{D''}$ and $\nu_{D'}$ are not easily related.

Nevertheless, note that $D$ and $D''$ are the same certificate since we only remove from $G$ an edge $e \notin D$. The only thing that can change by removing edge $e = (e^-, e^+)$ is that the right child of node $S'$ is no longer binary (i.e. we removed the last back edge). The question is if we can check quickly whether it is unary in $O(k)$ time: observe that $|\text{adj}(e^-)|$ is no longer the same, invalidating the flag $\text{is\_unary}$ (item (iv) of Section 3.3.1). Our idea is the following: instead of recomputing the certificate $D''$ in $O(k + \nu_{D''})$ time, we update the $\text{is\_unary}$ flag in just $O(k)$ time. We thus introduce a new operation $D'' = \text{unary}(D)$, a valid replacement for $D'' = \text{mdfs}_k(D)$ in $G - \{e\}$: it maintains the certificate while recomputing the flag $\text{is\_unary}$ in $O(k)$ time.

Lemma 3.17. Operation $\text{unary}(D)$ takes $O(k)$ time and correctly computes $D''$.

Proof. Consider the two certificates $D$ and $D''$: we want to compute directly $D''$ from $D$. Note that the only difference in the two certificate is the flag $\text{is\_unary}$. Hence, it suffices to show how to recompute $\text{is\_unary}$ from scratch in $O(k)$ time. This is equivalent to checking if the cutset $C(S)$ in $G - \{e\}$ contains only tree edges from $D$. It suffices to examine the each adjacency list $\text{adj}(u)$ for $u \in S$, until either we find an external or a back edge (so $\text{is\_unary}$ is false) or we scan all these lists (so $\text{is\_unary}$ is true): in the latter case, there are at most $2k$ edges in the adjacency lists of the vertices in $V[S]$ as these are the edges in $S$ or in $D$. All other edges are external or back edges.

Since there is no modification or impact on unary nodes of the recursion tree, we finalize the analysis.

Lemma 3.18. The cost of $\text{ListTrees}_v(S, D)$ on a binary node $S$ is $O(k + \nu_{D'} + |\text{comb}(S')|)$.

Proof. Consider the operations done in binary nodes. By Lemmas 3.15, 3.13 and 3.17 we have the following breakdown of the costs:

- Operation $e := \text{choose}(S, D)$ takes $O(k + \nu_{D'} + |\text{comb}(S')|)$;
3.4. OPTIMAL APPROACH: AMORTIZATION

- Operation $D' := \text{mdfs}_k(D)$ takes $O(k + \nu_{D'})$;
- Operation $D'' := \text{unary}(D)$ takes $O(k)$.

This sums up to $O(k + \nu_{D'} + |\text{comb}(S')|)$ time per binary node.

Lemma 3.19. The algorithm $\text{ListTrees}_{v_i}$ takes $O(s_i k)$ time and $O(mk)$ space.

Proof. By Lemma 3.18 we have a cost of $O(k + \nu_{D'} + |\text{comb}(S')|)$ per binary node on the recursion tree. Given that there are $O(s_i)$ binary nodes by Corollary 3.2 and given the amortization of $\nu_{D'}$ and $|\text{comb}(S')|$ in each left children $S'$ over the leafs of the recursion tree in Lemmas 3.14 and 3.16, the sums of $\sum S' k + \sum S' \nu_{D'} + \sum S' |\text{comb}(S')| = s_i k$. Additionally, by Lemma 3.9, the $O(s_i k)$ unary nodes give a total contribution $O(s_i k)$ to the cost. This is sums up to $O(s_i k)$ total time.

Note that our data structures are lists and array, so it is not difficult to replace them with persistent arrays and lists, a classical trick in data structures. As a result, we just need $O(1)$ space per pending recursive call, plus the space of the parking lists, which makes a total of $O(m)$ space.

Theorem 3.5. Algorithm $\text{ListTrees}_{v_i}$ takes a total of $O(sk)$ time, being therefore optimal, and $O(m)$ space.

Proof. Algorithm $\text{ListTrees}_{v_i}$ deletes the connected component of size smaller than $k$ and containing $v_i$ in the residual graph. So this cost sums up to $O(m)$. Otherwise, we can proceed as in the analysis of Algorithm $\text{ListTrees}_{v_i}$. Note that our data structures are lists and array, so it is not difficult to replace them with persistent arrays and lists, a classical trick in data structures. As a result, we just need $O(1)$ space per pending recursive call, plus the space of the parking lists, which makes a total of $O(m)$ space.

We finally show how to obtain an efficient delay. We exploit the following property on the recursion tree, which allows to associate a unique leaf with an internal node before exploring the subtree of that node (recall that we are in a recursion tree). Note that only the rightmost leaf descending from the root is not associated in this way, but we can easily handle this special case.

Lemma 3.20. For a binary node $S$ in the recursion tree, $\text{ListTrees}_{v_i}(S, D)$ outputs the $k$-subtree $D$ in the rightmost leaf descending from its left child $S'$.

Proof. Follow the rightmost spine in the subtree rooted at $S'$. An edge of $D$ is chosen after that all the external and back edges are deleted. Now, adding that edge to the partial solution, may bring new external and back edges. But they are again deleted along the path that follows the right spine. In other words, the path from $S'$ to its rightmost leaf includes only edges from $D$ when branching to the left, and removes the external and back edges when branching to the right.
Nakano and Uno [68] have introduced this nice trick. Classify a binary node $S$ in the recursion tree as even (resp., odd) if it has an even (resp., odd) number of ancestor nodes that are binary. Consider the simple modification to $\text{ListTrees}_{v_i}$ when $S$ is binary: if $S$ is even then output $D$ immediately before the two recursive calls; otherwise ($S$ is odd), output $D$ immediately after the two recursive calls.

**Theorem 3.6.** Algorithm 3 can be implemented with delay $t(k) = k^2$.

*Proof.* Between any two $k$-subtrees that are listed one after the other, we traverse $O(1)$ binary nodes and $O(k)$ unary nodes, so the total cost is $O(k^3)$ time in the worst case. \qed
Chapter 4

Listing k-subgraphs

When considering an undirected connected graph $G$ with $n$ vertices and $m$ edges, we solve the problem of listing all the connected induced subgraphs of $G$ with $k$ vertices. Figure 4.1 shows an example graph $G_1$ and its $k$-subgraphs when $k = 3$. We solve this problem optimally, in time proportional to the size of the input graph $G$ plus the size of the edges in the $k$-subgraphs to output.

There exist vast practical applications of using $k$-subgraphs as mining pattern in diverse fields. In general, determining subgraphs that appear frequently in one or more networks is considered relevant in the study of biological [1, 2, 45, 63], social [39, 40, 103] and other complex networks [3, 12, 15, 67]. These patterns, frequently called motifs, can provide insight to the function and design principles of these networks.

In the particular case of biological networks, Wernicke [105] has introduced the ESU algorithm as a base to find motifs. This algorithm solves the problem of listing the $k$-subgraphs of a network. The author further modifies the algorithm to sample $k$-subgraphs and is able to efficiently find network motifs. Nevertheless, a theoretical analysis of the algorithm is not performed and it does not seem to be possible to implement the algorithm in an optimal output-sensitive sensitive way.

Researchers focusing on finding motifs in graphs face the problem of counting or estimating the occurrences of each structure and thus face the graph isomorphism problem. For this reason, there appears to be little research done on efficiently listing the patterns themselves.

In [5], Avis and Fukuda propose the application of the general reverse search method and obtain an algorithm taking $O(mn\eta)$ time, where $\eta$ is the number of $k$-subgraphs in the input graph.

Equivalent to the problem of listing $k$-subgraphs is to list the vertices in the $k$-subgraphs of $G$. Although it is trivial to reduce one problem to the other, note that an optimal output-sensitive solution for the problem of listing $k$-subgraphs does not imply an optimal solution for the problem of listing its vertices, as in the latter case is sufficient to output the vertices in the $k$-subgraph (instead of the edges).
Interestingly, for the particular case of when graph $G$ is a tree, the problem of listing $k$-subgraphs is equivalent to the problem of listing $k$-subtrees. We achieve the same time bound as achieved in Chapter 3. For this particular case, the algorithm proposed in [102] is able to enumerate $k$-subgraphs in $O(1)$ per $k$-subgraph.

We present our solution based on the binary partition method (Section 2.4.1). We divide the problem of listing all the $k$-subgraphs in two subproblems by choosing a vertex $v \in V$: (i) we list the $k$-subgraphs that contain $v$, and (ii) those that do not contain $v$. We proceed recursively on these subproblems until there is just one $k$-subgraph to be listed. This method induces a binary recursion tree, and all the $k$-subgraphs are listed when reaching the leaves of the recursion tree.

As previously mentioned, in order to reach an output-sensitive algorithm, we maintain a certificate that allows us to determine efficiently if there exists at least one $k$-subgraph to be listed at any point in the recursion. Furthermore we select the vertex $v$, that divides the problem into two subproblems, in a way that facilitates the maintenance of the certificate.
4.1 Preliminaries

Given a graph $G = (V, E)$ and $V' \subseteq V$, $G[V'] = (V', E[V'])$ denotes the subgraph induced by the vertices $V'$. Note that we use the shorthand $E[V'] = \{(u, v) \in E \mid u, v \in V'\}$. A $k$-subgraph is a connected induced subgraph $G[V']$ such that $|V'| = k$ and $G[V']$ is connected. We denote by $S_k(G)$ the set of $k$-subgraphs in $G$. Formally:

$$S_k(G) = \{G[V'] \mid V' \subseteq V, |V'| = k \text{ and } G[V'] \text{ is connected} \}$$

For a vertex $u \in V$, $N(u)$ denotes the neighborhood of $u$. For a vertex set $S$, $N(S)$ is the union of neighborhoods of the vertices in $S$ while $N^*(S) = N(S) \setminus S$. Additionally, for a vertex set $S$, we define the cutset $C(S) \subseteq E$ such that $(u, v) \in C(S)$ if $u \in S$ and $v \in V[G] \setminus S$.

In this chapter we focus on the problem of efficiently listing $k$-subgraphs.

**Problem 4.1.** Given a simple undirected and connected graph $G = (V, E)$ and an integer $k$, list all subgraphs $G[V'] \in S_k(G)$.

An algorithm for this problem is said to be optimally output-sensitive if the time taken to solve Problem 4.1 is $O(m + \sum_{G[V'] \in S_k(G)} |E[V']|)$. Thus, an improvement of the result of Avis and Fukuda [5]. This algorithm is optimal in the sense that it takes time proportional to reading the input plus listing the edges in each of the induced subgraphs with $k$ vertices. Noting that the input graph $G$ is connected, we have that $O(m + \sum_{G[V'] \in S_k(G)} |E[V']|) = O(\sum_{G[V'] \in S_k(G)} |E[V']|)$ since each edge $e \in E$ belongs to at least one $k$-subgraph.

We dedicate the rest of the paper to proving the main result of the chapter.

**Theorem 4.2.** Problem 4.1 can be solved in $O(\sum_{G[V'] \in S_k(G)} |E[V']|)$ time.

4.2 Top-level algorithm

Given a graph $G = (V, E)$, we solve Problem 4.1 by using the natural approach of taking an ordering of the vertices $v_1, v_2, \ldots, v_n$ and listing all the connected induced subgraphs with $k$ vertices that include a vertex $v_i$ but do not include any vertex $v_j$ with $j < i$.

Additionally, we use a certificate $C$: an adequate data structure that, given a $k'$-subgraph $G[S]$ with $k' \leq k$ vertices, ensures that there exists a connected vertex set with $k$ vertices that includes $S$. We denote this relation as $S \subseteq C$. As a matter of convenience, we use $C$ to represent both the data structure and the connected set of vertices of size $k$. If there does not exist such a vertex set with size $k$, $C$ includes largest possible connected vertex set that includes $S$ (and thus $|C| < k$).

To efficiently implement this approach, we make use of the following operations:

- $\text{del}(u)$ deletes a vertex $u \in V$ and all its incident edges.
Algorithm 5 ListSubgraphs( $G = (V, E)$, $k$ )

1: for $i = 1, 2, \ldots, n - 1$ do
2: $S := \langle v_i \rangle$
3: $C := \text{certificate}(S)$
4: if $|C| = k$ then
5: ListSubgraphs$_{v_i}(S, C)$
6: del($v_i$)
7: else
8: for $u \in C$ do
9: del($u$)
10: end for
11: end if
12: end for

- undel($u$) restores $G$ to its previous state, before operation del($u$).
- certificate($S$) returns a certificate $C$ of $S$ (as defined above).

Let us now introduce Algorithm 5. We start by taking the vertices in $V$ in order (line 1) and, for each $v_i$ in $V$, we initialize an ordered set of nodes $S$ with $v_i$ (line 2) and compute its certificate $C$ (line 3). If $C$ contains $k$ vertices (lines 4-7) then there is at least a $k$-subgraph that includes $v_i$. In this case, we list all the $k$-subgraphs that include $v_i$ (line 5) and remove $v_i$ from $G$ (line 6) to avoid listing multiple times the same $k$-subgraph. If the certificate $C$ contains less than $k$ vertices (lines 7-11) then there is no $k$-subgraph that includes $v_i$. Furthermore, by the definition of $C$, there is no $k$-subgraph that includes each vertex $u \in C$ and thus $u$ can be deleted from $G$ (lines 8-10).

In the next section we present a recursive algorithm that, given an ordered set of vertices $S$ (initially $S = \langle v_i \rangle$) and a certificate $C$ of $S$, lists all the $k$-subgraphs that include $S$ and thus implements ListSubgraphs$_{v_i}(S, C)$ (line 5).

**Lemma 4.1.** Algorithm 5 lists each $k$-subgraphs in $G$ once and only once.

**Proof.** It follows directly from the definition of certificate that for each vertex $v_i \in V$, the algorithm calls ListSubgraphs$_{v_i}(\langle v_i \rangle, C)$ if $v_i$ belongs to at least one $k$-subgraph. On line 5, $v_i$ is removed and thus no other $k$-subgraph that includes $v_i$ is listed. Furthermore, if $v_i$ does not belong to any $k$-subgraph, then none of the vertices in its connected belong to a $k$-subgraph and thus can also be deleted (lines 8-10).

### 4.3 Recursion

Let us now focus on the problem of listing all the $k$-subgraphs that include a vertex $v_i$. In this section we use the follow operations as black boxes and we will detail
4.3. RECURSION

Algorithm 6 ListSubgraphs_{vi}(S, C)  

Invariant: $S$ is connected and $S \subseteq C$

1: if $|S| = k$ then
2: output($E[S]$)
3: return
4: end if
5: $v := \text{choose}(S)$
6: $I := \text{update\_left}(C, v)$
7: ListSubgraphs_{vi}(S ∪ {v}, C)
8: restore($C, I$)
9: del($v$)
10: $I := \text{update\_right}(C, v)$
11: if $|C| = k$ then
12: ListSubgraphs_{vi}(S, C)
13: end if
14: undel($v$)
15: restore($C, I$)

them in the next sections where their implementation is fundamental to the analysis
of the time complexity.

- \textbf{choose}(S, C) given a connected vertex set $S$ and certificate $C$, returns a vertex
  $v \in N^\ast(S)$. We will use the certificate $C$ to select a vertex $v$ that facilitates
  updating $C$.

- \textbf{update\_left}(C, v) updates the certificate $C$ such that $C$ is a certificate of
  $S \cup \{v\}$. Returns a data structure $I$ with the modifications made in $C$.

- \textbf{update\_right}(C, v) updates the certificate $C$ in the case of removal of $v$ from
  graph $G$. Note that, if $v$ does not belong to the certificate $C$, no modifications
  are needed. Returns a data structure $I$ with the modifications made in $C$.

- \textbf{restore}(C, I) restores the certificate $C$ to its previous state, prior to the
  modifications in $I$.

Algorithm 6 follows closely the binary partition method while maintaining the
invariant that $C$ is a certificate of $S$. This is of central importance to reach an
efficient algorithm since it allows us to avoid recursive calls that do not output any
$k$-subgraphs.

Given a connected vertex set $S$, we take a vertex $v \in N^\ast(S)$ (line 5). We then
perform \textit{leftward branching} (lines 6-8), recursively listing all the $k$-subgraphs in $G$
that include the vertices in $S$ and the vertex $v$. Note that by the definition of
\textbf{choose}(S), $v$ is connected to $S$ and thus we maintain the invariant that vertices in
$S$ are connected. Since $S$ is ordered, we define the operation $S \cup \{v\}$ to append
vertex \( v \) to the end of \( S \) and thus the vertices in \( S \) are kept in order of insertion.

We use the operation \texttt{update\_left}(\( S, v \)) (line 6) to maintain the invariant \( S \subseteq C \) in the recursive call (line 7). In line 8 we restore \( C \) to its state before the recursive call.

Successively, we perform \texttt{rightward\_branching} (lines 9-15), listing all \( k \)-subgraphs that include the vertices in \( S \) but do not include vertex \( v \). After removing \( v \) from the graph (line 9), we update the certificate \( C \) (line 10). If it was possible to update the certificate (i.e. \(| C | = k \)), we recursively list the \( k \)-subgraphs that include \( S \) in the residual graph \( G - \{ v \} \) (line 12). When returning from the recursive call, we restore the graph (line 14) and the certificate (line 15) to its previous state.

When the vertex set \( S \) has size \( k \) we reach the base case of the recursion (lines 1-4). Since \( S \) is connected, it induces a \( k \)-subgraph \( G[S] \) which we output (line 2). Note that in this case we do not proceed with the recursion (line 3).

\textbf{Lemma 4.2.} Given the existence of a certificate \( C \) of \( S \) and \( v = \texttt{choose}(S, C) \), there exists a \( k \)-subgraph that includes \( S \cup \{ v \} \) and thus a left branch always produces one or more \( k \)-subgraphs.

\textbf{Proof.} It follows from the definition of certificate that vertices in \( S \) are in a connected component of size at least \( k \). Noting that in a left branch we do not remove a vertex, \( S = S + \{ v \} \) is still in a connected component of size at least \( k \). Thus, the recursive call on line 7 will list at least one \( k \)-subgraph. \hfill \Box

\textbf{Lemma 4.3.} At each recursive call \texttt{ListSubgraphs}_{\( v \)}(\( S, C \)), we maintain the following invariants: (i) \( S \) is connected, (ii) \( C \) is a valid certificate of \( S \) and (iii) \texttt{ListSubgraphs}_{\( v \)}(\( S, C \)) is only invoked if there exists a \( k \)-subgraph that includes \( S \).

\textbf{Proof.} (i) By the definition of operation \texttt{choose}(\( S, C \)), \( v \in N^*(S) \) (line 5) and thus, on the recursive call of line 7 (corresponding to a left branch), \( S \cup \{ v \} \) is still connected. On the recursive call of line 12 (corresponding to a right branch), \( S \) is not altered. Thus we maintain the invariant that \( S \) is connected. (ii) Before both recursive calls of lines 5 and 7, the operations that maintain the certificate are called (lines 6 and 10). After returning from the recursion the certificate is restored to its previous state (lines 8 and 15) and thus the invariant \( S \subseteq C \) is maintained. (iii) By Lemma 4.2 on the recursive call of line 7 the invariant is trivially maintained. On the case of right branching, recursive call of line 12 we verify that \( S \) still belongs to a connected component of size at least \( k \) (line 11). \hfill \Box

\textbf{Lemma 4.4.} A call to \texttt{ListSubgraphs}_{\( v \)}(\( \langle v \rangle, C \)) lists the \( k \)-subgraphs that include \( v \).

\textbf{Proof.} A call to \texttt{ListSubgraphs}_{\( v \)}(\( S, C \)) recursively lists all \( k \)-subgraphs that include \( v \in N^*(S) \) (line 12). By Lemma 4.3 we maintain a certificate \( C \) that allows us to verify if there exist any \( k \)-subgraphs that include \( S \) in the residual graph \( G - \{ v \} \) (line 11). If that is the case, we invoke the right branch of the recursion (line 12).
When \(|S| = k\), we reach the base case of the recursion and \(S\) induces a \(k\)-subgraph that we output (line 2). Thus, we list all the \(k\)-subgraphs. Noting that we remove vertex \(v\) on the right branch of the recursion and that the set of \(k\)-subgraphs that include \(v\) is disjoint from the set of subgraphs that does not include \(v\), the same \(k\)-subgraph is not listed twice.

\[ S \mid v = k \] induces a \(k\)-subgraph that we output (line 2). Thus, we list all the \(k\)-subgraphs. Noting that we remove vertex \(v\) on the right branch of the recursion and that the set of \(k\)-subgraphs that include \(v\) is disjoint from the set of subgraphs that does not include \(v\), the same \(k\)-subgraph is not listed twice.

### 4.4 Amortization strategy

Let us consider the recursion tree \(R_i\) of \(\text{ListSubgraphs}_{v_i}(S, C)\). We denote a node \(r \in R_i\) by the arguments \(\langle S, C \rangle\) of \(\text{ListSubgraphs}_{v_i}(S, C)\). At the root \(r_i = \langle S, C \rangle\) of \(R_i\), \(S = \langle v_i \rangle\). A leaf node in the recursion tree corresponds to the base case of the recursion (and thus has no children). We say that an internal node \(r \in R\) is binary if and only if both the left and right branch recursive calls are executed. The node \(r\) is unary otherwise (by Lemma 4.2 left branching is always performed).

We propose a time cost charging scheme for each node \(r = \langle S, C \rangle \in R_i\). This cost scheme drives the definition of the certificate and implementation of the update operations.

\[
T(r) = \begin{cases} 
O(|E[S]|) & \text{if } r \text{ is leaf} \\
O(1) & \text{otherwise} 
\end{cases} \quad (4.1)
\]

From the correctness of the algorithm, it follows that the leaves of the recursion tree \(R_i\) are in one-to-one correspondence to the set of \(k\)-subgraphs that include \(v_i\). Additionally, note that \(E[S]\) are the edges in those \(k\)-subgraphs.

**Lemma 4.5.** The total time over all the nodes in the recursion trees \(R_1, \ldots, R_n\) given the charging scheme in Eq. 4.1 is \(O(\sum_{G[V'] \in S_k(G)} |E[V']|)\)

**Proof.** Since there is a one-to-one correspondence from the leaves in \(R_1, \ldots, R_n\) to the subgraphs in \(S_k(G)\) then the sum of the cost over all the leaves in the recursion tree is \(O(\sum_{G[V'] \in S_k(G)} |E[V']|)\). Furthermore, the total number of binary nodes in \(R\) is bounded by the number of leaves. Additionally, when considering any root to leaf path in \(R\) there are at most \(k\) left branches. This implies that there exist \(O(k|S_k(G)|)\) internal nodes in \(R\). Since any graph \(G[V'] \in S_k(G)\) is connected, \(|E[V']| \geq k - 1\) and \(O(k|S_k(G)|) + \sum_{G[V'] \in S_k(G)} |E[V']|) = O(\sum_{G[V'] \in S_k(G)} |E[V']|)\) equals the cost of the operations on the graph and the certificate. Although, on internal nodes of \(R\), we are not able to reach \(O(1)\) time we will show that the time cost can be amortized according to Eq. 4.1.

### 4.5 Certificate

A certificate \(C\) of a \(k'\)-subgraph \(G[S]\) is a data structure that uses a truncated multi-source depth-first-search tree and classifies a vertex \(v \in N^+(S)\) as internal (to the
certificate $C$) or external. We say that a vertex $v \in N^*(S)$ is internal if $v \in C$ and external otherwise.

Given a $k'$-subgraph $G[S]$ with $S = v_1, v_2, \ldots, v_{k'}$, the multi-source DFS tree $C$ contains the vertices in $S$, which are conceptually treated as a collapsed vertex with the ordered cutlist $\hat{C}(S)$ as adjacency list. Recall that $\hat{C}(S)$ contains the edges in the cutset $C(S)$ ordered by the rank of their endpoints in $S$. If two edges have the same endpoint vertex $v \in S$, we use the order in which they appear in the adjacency list of $v$ to break the tie. Also, all the vertices in $S$ are conceptually marked as visited at the beginning of the DFS, so $u_j$ is never part of the DFS tree starting from $u_i$ for any two distinct $u_i, u_j \in S$. Hence the adopted terminology of multi-source.

The multi-source DFS tree is truncated when it reaches $k$ vertices. In this case we say that $|C| = k$ and know that there exists a $k$-subgraph that includes all vertices in $S$. This is the main goal of maintaining the certificate.

The data structure that implements the certificate $C$ of $S$ is a partition $C = F \cup B$ where $F$ is a forest storing the edges of the multi-source DFS whose both endpoints are in $V[C] - V[S]$. It represents the remaining forest of the multi-source DFS tree when the collapsed vertices in $S$ are removed. Moreover, $B$ is a vector that allows to efficiently test if a vertex $v$ belongs to $C$.

(i) We store the forest as a sorted doubly-linked list of the roots of the trees in the forest. The order of this list is that induced by $\hat{C}(S)$: a root $r$ precedes a root $t$ if the edge in $(x, r) \in \hat{C}(S)$ precedes $(y, t) \in \hat{C}(S)$. We also keep a pointer to the last leaf, in the order of the multi-source DFS visit, of the last tree in $F$.

(ii) We maintain a vector $B$ where $B[v] = \text{true}$ if and only if $v \in C$.

Furthermore, the certificate implements the following operations:

- Let $r$ be the root of the last tree in $F$. Operation $\text{promote}(C)$ removes $r$ from $F$ and appends the children of $r$ to $F$ (and thus the children of $r$ become roots of trees in $F$).

- Let $l$ be the last leaf of the last tree $T$ in $F$ (in the order of the DFS visit). Operation $\text{removelastleaf}(C)$ removes $l$ from $T$. Note that $l$ can also be the last root in $F$, in this case this root is removed since $T$ becomes empty.

**Lemma 4.6.** Operations $\text{promote}(C)$ and $\text{removelastleaf}(C)$ can be performed in $O(1)$ time.

**Proof.** Operation $\text{promote}(C)$. By maintaining the trees in $F$ using the classic first-child next-children encoding of trees, it is possible to take the linked list of children of $r$ and append it to $F$ in constant time. Operation $\text{removelastleaf}(C)$ can also be implemented in constant time since we maintain a pointer to the last leaf $l$ and only have to remove it from the certificate. 

\qed
As we will show in Section 4.5.1, this implementation of the certificate allows it to be efficiently updated. Additionally, we show that it can be computed in time proportional to the number of edges in the certificate $C$.

Lemma 4.7. Operation $C = \text{certificate}(\langle v_i \rangle)$ can be performed in $O(|E[C]|)$ time.

Proof. We start performing a classic DFS visit starting from $v_i$ and adding the tree edges visited to $F$. When $k$ vertices are collected, we stop the visit. On vertex $x$, we visit edges $e = (x,y)$ that either collect a vertex $y$ not previously visited and we set $B[y] = \text{true}$ or connect $x$ to a vertex $y$ already visited ($e$ is a back edge). In both cases $e \in E[C]$ and we do not visit such edge $e$ more than twice (once of each endpoint).

Also with the goal of efficiently updating the certificate, we define a specialization of the operation $v = \text{choose}(S, C)$ (illustrated in Figure 4.2). Recall that the only requirement for the correctness of the algorithm is that $v \in N^*(S)$. By using the classification of $v$ as internal or external, we are able to select a vertex that allows efficiently updating the certificate.

- $\text{choose}(S, C)$ returns a vertex $v \in N^*(S)$ such that: $v$ is external if such vertex $v \notin C$ exists; otherwise return the last root of $F$ (in this case $v$ is internal).

### 4.5.1 Maintaining the certificate

Let us now show how to maintain the certificate in the time bounds defined in Eq. 4.1. In the case of operation $\text{update.left}(C, v)$ with $v = \text{choose}(S, C)$, we can
update the certificate for the left branch in constant time. When \( v \) is an external vertex, it is sufficient to remove the last leaf (in DFS order) of the last tree of \( F \) while appending vertex \( v \) to \( S \). If \( v \) is internal, and thus already part of \( C \), the vertices in the certificate remain the same and it is enough to update the data structure that represents \( C \).

**Lemma 4.8.** Operation \( I = \text{update}\_\text{left}(C, v) \) can be performed in \( O(1) \) time.

**Proof.** Given a certificate \( C \) of \( S \), it is possible to update \( C \) so that \( C \) becomes the certificate of \( S \cup \{v\} \) in \( O(1) \) time. If \( v \) is an external vertex (i.e. \( v \notin C \)), when we append \( v \) to \( S \), the data structure \( C \) is still valid but has one vertex too many (i.e. it is a certificate that there exists a \( k + 1 \)-subgraph) and thus we call \( \text{removelastleaf}(C) \) to decrease the size by one while maintaining the property that \( C \) is still a multi-source DFS tree. In the case that \( v \) is an internal vertex, the vertices in the certificate \( C \) are still the same but we have to update \( F \) to reflect that the size of \( V[S+\{v\}]−V[C] \) is now smaller. Since \( v \) is the root of the last tree in \( F \) it is enough to call operation \( \text{promote}(C) \) while maintaining the invariant that \( C \) is a multi-source truncated DFS from \( S \). The modifications to \( C \) are stored in \( I \) in order to restore the certificate \( C \) to its previous state using operation \( \text{restore}(C, I) \).

Performing the operation \( \text{update}\_\text{right}(C, v) \) when \( v \) is external to the certificate is also easy as the certificate \( C \) remains valid after the removal of vertex \( v \).

**Lemma 4.9.** Operation \( I = \text{update}\_\text{right}(C, v) \) can be performed in \( O(1) \) time when \( v \) is external.

**Proof.** Since \( v \notin C \), removing the vertex \( v \) has no impact in the certificate and therefore \( C \) is still a valid certificate of \( S \) in the graph \( G − \{v\} \). In this case \( I \) is empty.

To perform the operation \( \text{update}\_\text{right}(C, v) \) when \( v \) is internal is more complex since we are removing from the graph \( G \) a vertex that is part of the certificate. By the definition of operation \( \text{choose}(C) \), this operation invalidates the last tree of the forest \( F \) (other trees have been completely explored by the DFS and, since the graph is undirected, there are no cross edges between the trees in \( F \)). Nevertheless, noting that there are no vertices in \( N^+(S) \) that do not belong to the certificate, we prove that we can recompute part of the multi-source DFS tree in the budget defined in Eq. 4.1. When this recomputation is able to update the certificate \( C \) with \( |C| = k \), we prove that we can charge its cost on the leaf that outputs the \( k \)-subgraph \( G[C] \). When the recomputation leads to a \( C \) with \( |C| < k \), we prove that there still exists a leaf where we can charge the cost of recomputing the certificate. Furthermore, we prove that the same leaf is not charged twice and thus we respect the cost scheme defined in Eq. 4.1.

**Lemma 4.10.** Operation \( I = \text{update}\_\text{right}(C, v) \), when \( v \) is internal, can be performed in the time defined by Eq. 4.1.
Proof. In this operation we have removed vertex \( v \) which, by definition of choose(\( C \)), is the last root of the last tree \( T \) of \( F \). We note that, when removing \( v \), the tree \( T \) has been split into several branches \( T'_1, T'_2, \ldots, T'_i \), one for each of the \( i \) children of \( v \) in \( T \). Additionally, by the definition of choose(\( C \)), we remark that there are no edges \( e = (x, y) \) where \( x \in S \) and \( y \notin V[C] - V[S] \) and thus every edge leaving from a vertex in \( S \) leads to a vertex in the forest. Furthermore, by the properties of the multi-source DFS tree, we know that the trees \( T'_1, \ldots, T'_{i-1} \) are closed while the tree \( T'_i \) is possibly open (i.e. can lead to vertices not previously in the certificate).

We update the certificate by performing the multi-source DFS tree much like in Lemma 4.7 but with small twist. (i) If we reach a vertex in a tree of \( F \) different from \( T \) we reuse the portion of the DFS visit already performed. This tree of \( F \) is still valid as there exist no cross edges between tree (a key property and the main reason why we use a DFS). Thus, we proceed w.l.o.g. assuming that \( F \) is composed only by its last tree \( T \); (ii) If we reach a vertex in the trees \( T'_1, \ldots, T'_i \) we keep performing the visit until \( k \) vertices are visited or we have no edges left to explore. The modifications performed are kept in \( I \) and when a vertex \( x \) is included in (resp. removed from) the certificate \( B[x] \) is set to true (resp. false).

The existence of a certificate \( C \) with \( |C| = k \), depends of how many of the trees \( T'_1, \ldots, T'_{i-1} \) we are able to “recapture”. Only if we are able to reach the tree \( T'_i \), we will be able to include extra vertices not previously in the certificate (to account for the removal of \( v \) and possible the disconnection of \( T'_1, \ldots, T'_{i-1} \)). For the purposes of this proof, let us consider two different cases: (a) when the visit reaches \( k \) vertices and, (b) when the visit does not reach \( k \) vertices. Note that in case (b), the right branch of the recursion tree will not be performed.

![Figure 4.3: Case (a) of update_right(\( C, v \)) when \( v \) is internal](image-url)
(a) Since the updated certificate $C$ includes $k$ vertices and noting that we did not explore edges external to $C$, this operation takes $O(|E[C]|)$ time. In this case we perform a rightward branch. We charge this cost to the leaf corresponding to the output of $k$-subgraph $G[C]$. Given the order choose($C$) selects the vertices to recurse, the path in the recursion tree $R$ from the current node of recursion $r_x \in R$ to the leaf corresponding to the output of $C$ is a sequences of left branches with $v$ being an internal vertex interleaved with right branches with $v$ being an external vertex. Noting that from $r_x$ to this leaf we do not perform any right branch where $v$ is internal, any leaf is charged at most once. This case is illustrated in Figure 4.3.

(b) When $|C| < k$ the visit still takes $O(|E[C]|)$ time but, since $G[C]$ is not a $k$-subgraph, we are not able to amortize it in the same way as in (a). Let us assume that the vertices $\alpha$ of the trees $T'_1, \ldots T'_i$ were “recaptured”, the vertices $\beta$ of the trees $T'_1, \ldots T'_i$ were not “recaptured” and that the vertices $\gamma$ not previously in $C$ were visited. Note that $|\alpha| + |\gamma| + |S| < k$. We are able to charge the $O(|E[C]|)$ cost to a leaf (and thus $k$-subgraph) $X$ that includes every edge in $E[C]$. The $k$-subgraph $X$ is formed in the following way: (i) $X$ includes the vertices in $S$; (ii) $X$ includes the vertex $v$; (iii) includes the vertices in $\alpha$ and $\gamma$; (iv) Since $v$ is part of $X$, all the vertices in $\beta$ are connected to the vertices in $X$ and thus we include a sufficient number of those in order to have $|X| = k$. Note that $E[C] \subset E[X]$. When $\gamma \neq \emptyset$, the leaf corresponding to $X$ is in a right branch of one of the descendants of the left branch of the current node of the recursion and thus is not charged more than once. The limit case when $\beta = \emptyset$ implies that no new nodes were visited in the update of

Figure 4.4: Case (b) of update_right($C, v$) when $v$ is internal
the certificate (i.e. $\gamma = \emptyset$) and thus the certificate $C$ previous to the update is the only leaf in the current branch of the recursion. In this limit case is easy to avoid recomputing the certificate in any descendant. Therefore, the cost of operation $\text{update} _\text{right}(C, v)$ can be charged on the leaves of the recursion according to the cost defined in Eq. 4.1.

**Lemma 4.11.** Operation $\text{restore}(C, I)$, can be performed in the time defined by Eq. 4.7.

**Proof.** We use standard data structures (i.e. linked lists) for the representation of certificate $C$. There exist persistent versions of these data structures that maintain a stack of modifications applied to them and that can restore its contents to their previous states. Given the modifications in $I$, these data structures take $O(|I|)$ time to restore the previous version of $C$. Given that the time taken to fill the modification in $I$ done on operations $\text{update} _\text{left}(C, v)$ and $\text{update} _\text{right}(C, v)$ respects Eq. 4.1, this implies that operation $\text{restore}(C, I)$ can be performed in the same time bound.

### 4.6 Other operations

In order to implement operation $\text{choose}(S, C)$ within the cost established in Eq. 4.1 we have to avoid visiting the edges which are internal to the certificate $C$ multiple times. The naive approach of finding the first edge that connects the vertices in $S$ to an external vertex can take up to $O(k^2)$ time, as we possibly have to visit all edges internal to $C$.

Let us consider the set of nodes $N$ in the recursion tree $R_i$ that have the same vertices in the certificate as the node $r = (S, C) \in R_i$. The recursion nodes in $N$ correspond to a sequence of right branches in external vertices and left branches on internal vertices.

We make use of a data structure which we call parking lists, where we place an edge $e$ internal to certificate $C$ when visiting it for the first time. Then, we remove $e$ from the graph $G$ to avoid visiting it additional times. This allows us, for the recursion nodes in $N$ whose vertices in the certificate $C$ are the same, to only visit each edge in $E[C]$ once. We then prove that this cost can be amortized according to Eq. 4.1. The parking lists are a set of lists $P_0, P_1, \ldots, P_n$ corresponding to each $v_i \in V$. When implementing operation $\text{choose}(S, C)$, if we visit an edge $e = (x, y)$ from the graph $G$ such that $x, y \in C$ we append $e$ to $P_x$ and remove it from $\text{adj}(x)$. Note that, for the purpose of simplification, this modification is restricted to operation $\text{choose}(S, C)$ and for other operations, such as the ones necessary to maintain the certificate, internal edges are still present (i.e. the adjacency list of vertex $u \in V$ is $\text{adj}(u) \cup P_u$).
Lemma 4.12. Operation choose\((S, C)\) can be performed within the time cost defined in Eq 4.1.

Proof. For each vertex \(x \in S\) we take each edge \(e = (x, y) \in \text{adj}(x)\). When vertex \(y \notin C\), we return \(y\). This can be tested in constant time by checking if vector \(B[y] = \text{false}\). When all edges are visited without finding such vertex \(y\), we return the root of the last tree in forest \(F\).

During this process, when we take and edge \(e = (x, y)\) with \(y \in C\), we remove \(e\) from \(\text{adj}(x)\) and set \(P_x := P_x \cup \{e\}\). Furthermore, on the implementation of operation \text{removelastleaf}(C), when removing vertex \(l\) from the certificate we update \(\text{adj}(l) = \text{adj}(l) \cup P_x\). This operation can be done in constant time.

Consider the set of nodes \(N\), forming a path in the recursion tree \(R_i\), where the vertices in their certificates \(C\) are the same. By using this strategy, we do not visit any edge in \(E[C]\) more than twice (once for each endpoint) and we can charge this cost to the leaf corresponding to the output of \(C\). Since there is only one such path for the set of vertices in certificate \(C\), the cost of this operation can be charged according to Eq. 4.1. We are now left with the analysis of operations \text{del}(v)\) and \text{undel}(v)\) in a node \(\langle S, C \rangle\) of the recursion tree \(R_i\). One thing to note is that the adjacency list \(\text{adj}(v)\) can have up to \(n - 1\) edges and, at first sight, we cannot afford to pay this cost. A key insight is that each of the edges in \(\text{adj}(v)\) is connected to the certificate \(C\) and thus it implies that \(e\) is part of some \(k\)-subgraphs. We now prove that we can amortize the cost of \text{del}(v)\) in those subgraphs.

Lemma 4.13. Operations \text{del}(v)\) and \text{undel}(v)\) can be performed within the time cost defined in Eq 4.1.

Proof. Consider the deletion of each edge incident in \(v\), \(e = (v, w) \in \text{adj}(v)\). Note that any vertex \(w\) is connected to \(S\) though \(v\) and thus there will be internal nodes \(\langle S', C' \rangle\) of the recursion tree with \(S' = S \cup \{v\} \cup \{w\}\). We charge the \(O(1)\) cost of removing edge \(e\) on the first internal node \(\langle S', C' \rangle \in R_i\). Since, for a given \(S\) we remove \((v, w)\) only once, this internal node is not charged multiple times. Operation \text{undel}(v)\) can be performed in the same time bound.

Lemma 4.14. We can maintain the parking lists in such way that they include every edge internal to the certificate within the time cost defined in Eq 4.1.

Proof. When we add one external vertex \(v\) to the certificate, we can visit every edge incident in \(v\) and add it to \(P_v\) if it is internal to the certificate. By a similar argument to the one used in Lemma 4.13, every edge \(e = (v, w) \in \text{adj}(v)\) can be visited. We can charge \(O(1)\) cost of visiting \(e\) to the first internal node \(\langle S', C' \rangle \in R_i\) where internal node \(\langle S', C' \rangle\) is the first node of the recursion tree with \(S' = S \cup \{v\} \cup \{w\}\). Since \(v\) is external and \(v\) is removed on the right branch of the recursion, the edges incident in \(v\) are not visited more than once. When \(v\) is internal, every edge internal to the certificate has already been discovered by operation choose\((C)\).
Having analyzed each operation performed in a node of the recursion tree, we are able to prove the following lemma.

**Lemma 4.15.** The operations a node \( r \) of the recursion tree \( R \) can be performed in the time defined in Eq. 4.1:

\[
T(r) = \begin{cases} 
O(|E[S]|) & \text{if } r \text{ is leaf} \\
O(1) & \text{otherwise}
\end{cases}
\]

**Proof.** Directly from Lemmas 4.12, 4.8, 4.9, 4.10, 4.13 and 4.11 each operation performed on an internal node of the recursion tree can be performed within the time cost defined. Furthermore, on the leaves of the recursion tree, we only have to output the edges on \( E[S] \), which can be recorded in the parking lists by Lemma 4.14.

With Lemma 4.15 and Lemma 4.5 we are able to prove Lemma 4.16 which completes the proof of Theorem 4.1.

**Lemma 4.16.** Algorithm 5 lists all \( k \)-subgraphs in \( O(\sum_{G[V'] \in S_k(G)} |E[V']|) \)

**Proof.** By Lemmas 4.5 and 4.15, the total time spent in \( \text{ListSubgraphs}_{k}(S,C) \) (line 5) is \( O(\sum_{G[V'] \in S_k(G)} |E[V']|) \). By Lemma 4.7, the cost of each call to \( \text{certificate}(S) \) (line 3) is \( O(|E[C]|) \). In the case \( |C| = k \) (lines 4-6), this cost can be amortized in the leaf corresponding to the output of \( k \)-subgraph \( C \). When \( |C| < k \) (lines 7-10), we remove every vertex in \( C \) and incident edges, thus the total cost over these operations is \( O(m) \). Noting that the input graph \( G \) is connected, each edge belongs to at least a \( k \)-subgraph and the total time taken by Algorithm 5 is \( O(\sum_{G[V'] \in S_k(G)} |E[V']|) \)
Chapter 5

Listing cycles and st-paths

Listing all the simple cycles (hereafter just called cycles) in a graph is a classical problem whose efficient solutions date back to the early 70s. For a graph with \( n \) vertices and \( m \) edges, containing \( \eta \) cycles, the best known solution in the literature is given by Johnson's algorithm \[43\] and takes \( O((\eta + 1)(m + n)) \) time. This solution is surprisingly not optimal for undirected graphs: to the best of our knowledge, no theoretically faster solutions have been proposed in almost 40 years.

Results

Originally introduced in \[10\], we present the first optimal solution to list all the cycles in an undirected graph \( G \). Specifically, let \( \mathcal{C}(G) \) denote the set of all these cycles (\(|\mathcal{C}(G)| = \eta\)). For a cycle \( c \in \mathcal{C}(G) \), let \(|c|\) denote the number of edges in \( c \). Our algorithm requires \( O(m + \sum_{c \in \mathcal{C}(G)} |c|) \) time and is asymptotically optimal: indeed, \( \Omega(m) \) time is necessarily required to read \( G \) as input, and \( \Omega(\sum_{c \in \mathcal{C}(G)} |c|) \) time is necessarily required to list the output. Since \(|c| \leq n\), the cost of our algorithm never exceeds \( O(m + (\eta + 1)n) \) time.

Along the same lines, we also present the first optimal solution to list all the simple paths from \( s \) to \( t \) (shortly, st-paths) in an undirected graph \( G \). Let \( \mathcal{P}_{st}(G) \) denote the set of st-paths in \( G \) and, for an st-path \( \pi \in \mathcal{P}_{st}(G) \), let \(|\pi|\) be the number of edges in \( \pi \). Our algorithm lists all the st-paths in \( G \) optimally in \( O(m + \sum_{\pi \in \mathcal{P}_{st}(G)} |\pi|) \) time, observing that \( \Omega(\sum_{\pi \in \mathcal{P}_{st}(G)} |\pi|) \) time is necessarily required to list the output.

We prove the following reduction to relate \( \mathcal{C}(G) \) and \( \mathcal{P}_{st}(G) \) for some suitable choices of vertices \( s, t \): if there exists an optimal algorithm to list the st-paths in \( G \), then there exists an optimal algorithm to list the cycles in \( G \). Hence, we can focus on listing st-paths.
CHAPTER 5. LISTING CYCLES AND ST-PATHS

(a) Graph $G_1$
(b) $st$-paths of $G_1$
(c) Cycles of $G_1$

Figure 5.1: Example graph $G_1$, its $st$-paths and cycles

Previous work

The classical problem of listing all the cycles of a graph has been extensively studied for its many applications in several fields, ranging from the mechanical analysis of chemical structures [85] to the design and analysis of reliable communication networks, and the graph isomorphism problem [104]. In particular, at the turn of the seventies several algorithms for enumerating all cycles of an undirected graph have been proposed. There is a vast body of work, and the majority of the algorithms listing all the cycles can be divided into the following three classes (see [8, 57] for excellent surveys).

1. **Search space algorithms.** According to this approach, cycles are looked for in an appropriate search space. In the case of undirected graphs, the cycle vector space [16] turned out to be the most promising choice: from a basis for this space, all vectors are computed and it is tested whether they are a cycle. Since the algorithm introduced in [104], many algorithms have been proposed: however, the complexity of these algorithms turns out to be exponential in the dimension of the vector space, and thus in $n$. For planar graphs, an algorithm listing cycles in $O((\eta + 1)n)$ time was presented in [86].

2. **Backtrack algorithms.** By this approach, all paths are generated by backtrack and, for each path, it is tested whether it is a cycle. One of the first algorithms is the one proposed in [90], which is however exponential in $\eta$. By adding a simple pruning strategy, this algorithm has been successively modified in [89]: it lists all the cycles in $O(nm(\eta + 1))$ time. Further improvements were proposed in [43, 87, 73], leading to $O((\eta + 1)(m + n))$-time algorithms that work for both directed and undirected graphs.
3. **Using the powers of the adjacency matrix.** This approach uses the so-called *variable adjacency matrix*, that is, the formal sum of edges joining two vertices. A non-zero element of the \( p \)-th power of this matrix is the sum of all walks of length \( p \): hence, to compute all cycles, we compute the \( n \)th power of the variable adjacency matrix. This approach is not very efficient because of the non-simple walks. Algorithms based on this approach (e.g. [71, 108]) basically differ only on the way they avoid to consider walks that are neither paths nor cycles.

Almost 40 years after Johnson’s algorithm [43], the problem of efficiently listing all cycles of a graph is still an active area of research (e.g. [9, 36, 41, 55, 77, 108, 79]). New application areas have emerged in the last decade, such as bioinformatics: for example, two algorithms for this problem have been proposed in [47, 48] while studying biological interaction graphs. Nevertheless, no significant improvement has been obtained from the theory standpoint: in particular, Johnson’s algorithm is still the theoretically most efficient. His \( O((\eta + 1)(m + n)) \)-time solution is surprisingly not optimal for undirected graphs as we show in this chapter.

**Difficult graphs for Johnson’s algorithm**

It is worth observing that the analysis of the time complexity of Johnson’s algorithm is not pessimistic and cannot match the one of our algorithm for listing cycles. For example, consider the sparse “diamond” graph \( D_n = (V, E) \) in Fig. 5.2 with \( n = 2k + 3 \) vertices in \( V = \{ a, b, c, v_1, \ldots, v_k, u_1, \ldots, u_k \} \). There are \( m = \Theta(n) \) edges in \( E = \{(a, c), (a, v_i), (v_i, b), (b, u_j), (u_i, c) \} \) for \( 1 \leq i \leq k \}, \) and three kinds of (simple) cycles: (1) \((a, v_i), (v_i, b), (b, u_j), (u_j, c), (c, a) \) for \( 1 \leq i < j \leq k \); (2) \((a, v_i), (v_i, b), (b, v_j), (v_j, a) \) for \( 1 \leq i < j \leq k \); (3) \((b, u_i), (u_i, c), (c, u_j), (u_j, b) \) for \( 1 \leq i < j \leq k \), totaling \( \eta = \Theta(n^2) \) cycles. Our algorithm takes \( \Theta(n + k^2) = \Theta(\eta) = \Theta(n^2) \) time to list these cycles. On the other hand, Johnson’s algorithm takes \( \Theta(n^3) \) time, and the discovery of the \( \Theta(n^2) \) cycles in (1) costs \( \Theta(k) = \Theta(n) \) time each: the backtracking procedure in Johnson’s algorithm starting at \( a \), and passing through \( v_i, b \) and \( u_j \) for some \( i, j \), arrives at \( c \): at that point, it explores all the vertices \( u_l \) \( (l \neq i) \) even if they do not lead to cycles when coupled with \( a, v_i, b, u_j, \) and \( c \).
5.1 Preliminaries

Let $G = (V, E)$ be an undirected connected graph with $n = |V|$ vertices and $m = |E|$ edges, without self-loops or parallel edges. For a vertex $u \in V$, we denote by $N(u)$ the neighborhood of $u$ and by $d(u) = |N(u)|$ its degree. $G[V']$ denotes the subgraph induced by $V' \subseteq V$, and $G - u$ is the induced subgraph $G[V \setminus \{u\}]$ for $u \in V$. Likewise for edge $e \in E$, we adopt the notation $G - e = (V, E \setminus \{e\})$. For a vertex $v \in V$, the postorder DFS number of $v$ is the relative time in which $v$ was last visited in a DFS traversal, i.e. the position of $v$ in the vertex list ordered by the last visiting time of each vertex in the DFS.

Paths are simple in $G$ by definition: we refer to a path $\pi$ by its natural sequence of vertices or edges. A path $\pi$ from $s$ to $t$, or $st$-path, is denoted by $\pi = s \leadsto t$. Additionally, $\mathcal{P}(G)$ is the set of all paths in $G$ and $\mathcal{P}_{s,t}(G)$ is the set of all $st$-paths in $G$. When $s = t$ we have cycles, and $\mathcal{C}(G)$ denotes the set of all cycles in $G$. We denote the number of edges in a path $\pi$ by $|\pi|$ and in a cycle $c$ by $|c|$. In this chapter, we consider the following problems.

**Problem 5.1** (Listing st-Paths). Given an undirected graph $G = (V, E)$ and two distinct vertices $s, t \in V$, output all the paths $\pi \in \mathcal{P}_{s,t}(G)$.

**Problem 5.2** (Listing Cycles). Given an undirected graph $G = (V, E)$, output all the cycles $c \in \mathcal{C}(G)$.

Our algorithms assume without loss of generality that the input graph $G$ is connected, hence $m \geq n - 1$, and use the decomposition of $G$ into biconnected components. Recall that an articulation point (or cut-vertex) is a vertex $u \in V$ such that the number of connected components in $G$ increases when $u$ is removed. $G$ is biconnected if it has no articulation points. Otherwise, $G$ can always be decomposed into a tree of biconnected components, called the block tree, where each biconnected component is a maximal biconnected subgraph of $G$ (see Fig. 5.3), and two biconnected components are adjacent if and only if they share an articulation point.
5.2 Overview and main ideas

While the basic approach is simple (see the binary partition in point 3), we use a number of non-trivial ideas to obtain our optimal algorithm for an undirected (connected) graph $G$ as summarized in the steps below.

1. Prove the following reduction. If there exists an optimal algorithm to list the $st$-paths in $G$, there exists an optimal algorithm to list the cycles in $G$. This relates $C(G)$ and $P_{st}(G)$ for some choices $s,t$.

2. Focus on listing the $st$-paths. Consider the decomposition of the graph into biconnected components (bccs), thus forming a tree $T$ where two bccs are adjacent in $T$ iff they share an articulation point. Exploit (and prove) the property that if $s$ and $t$ belong to distinct bccs, then (i) there is a unique sequence $B_{s,t}$ of adjacent bccs in $T$ through which each $st$-path must necessarily pass, and (ii) each $st$-path is the concatenation of paths connecting the articulation points of these bccs in $B_{s,t}$.

3. Recursively list the $st$-paths in $B_{s,t}$ using the classical binary partition (i.e. given an edge $e$ in $G$, list all the cycles containing $e$, and then all the cycles not containing $e$): now it suffices to work on the first bcc in $B_{s,t}$, and efficiently maintain it when deleting an edge $e$, as required by the binary partition.

4. Use a notion of certificate to avoid recursive calls (in the binary partition) that do not list new $st$-paths. This certificate is maintained dynamically as a data structure representing the first bcc in $B_{s,t}$, which guarantees that there exists at least one new solution in the current $B_{s,t}$.

5. Consider the binary recursion tree corresponding to the binary partition. Divide this tree into spines: a spine corresponds to the recursive calls generated by the edges $e$ belonging to the same adjacency list in $B_{s,t}$. The amortized cost for each listed $st$-path $\pi$ is $O(|\pi|)$ when there is a guarantee that the amortized cost in each spine $S$ is $O(\mu)$, where $\mu$ is a lower bound on the number of $st$-paths that will be listed from the recursive calls belonging to $S$. The (unknown) parameter $\mu$, which is different for each spine $S$, and the corresponding cost $O(\mu)$, will drive the design of the proposed algorithms.

5.2.1 Reduction to $st$-paths

We now show that listing cycles reduces to listing $st$-paths while preserving the optimal complexity.

**Lemma 5.1.** Given an algorithm that solves Problem 5.1 in $O(m + \sum_{\pi \in P_{s,t}(G)} |\pi|)$ time, there exists an algorithm that solves Problem 5.2 in $O(m + \sum_{c \in C(G)} |c|)$ time.
**Proof.** Compute the biconnected components of $G$ and keep them in a list $L$. Each (simple) cycle is contained in one of the biconnected components and therefore we can treat each biconnected component individually as follows. While $L$ is not empty, extract a biconnected component $B = (V_B, E_B)$ from $L$ and repeat the following three steps: (i) compute a DFS traversal of $B$ and take any back edge $b = (s, t)$ in $B$; (ii) list all $st$-paths in $B - b$, i.e. the cycles in $B$ that include edge $b$; (iii) remove edge $b$ from $B$, compute the new biconnected components thus created by removing edge $b$, and append them to $L$. When $L$ becomes empty, all the cycles in $G$ have been listed.

Creating $L$ takes $O(m)$ time. For every $B \in L$, steps (i) and (iii) take $O(|E_B|)$ time. Note that step (ii) always outputs distinct cycles in $B$ (i.e. $st$-paths in $B - b$) in $O(|E_B| + \sum_{\pi \in P_{s,t}(B-b)} |\pi|)$ time. However, $B - b$ is then decomposed into biconnected components whose edges are traversed again. We can pay for the latter cost: for any edge $e \neq b$ in a biconnected component $B$, there is always a cycle in $B$ that contains both $b$ and $e$ (i.e. it is an $st$-path in $B - b$), hence $\sum_{\pi \in P_{s,t}(B-b)} |\pi|$ dominates the term $|E_B|$, i.e. $\sum_{\pi \in P_{s,t}(B-b)} |\pi| = \Omega(|E_B|)$. Therefore steps (i)–(iii) take $O(\sum_{\pi \in P_{s,t}(B-b)} |\pi|)$ time. When $L$ becomes empty, the whole task has taken $O(m + \sum_{e \in E(G)} |c|)$ time. 

### 5.2.2 Decomposition in biconnected components

We now focus on listing $st$-paths (Problem 5.1). We use the decomposition of $G$ into a block tree of biconnected components. Given vertices $s, t$, define its bead string, denoted by $B_{s,t}$, as the unique sequence of one or more adjacent biconnected components (the beads) in the block tree, such that the first one contains $s$ and the last one contains $t$ (see Fig. 5.3): these biconnected components are connected through articulation points, which must belong to all the paths to be listed.

**Lemma 5.2.** All the $st$-paths in $P_{s,t}(G)$ are contained in the induced subgraph $G[B_{s,t}]$ for the bead string $B_{s,t}$. Moreover, all the articulation points in $G[B_{s,t}]$ are traversed by each of these paths.

**Proof.** Consider an edge $e = (u, v)$ in $G$ such that $u \in B_{s,t}$ and $v \notin B_{s,t}$. Since the biconnected components of a graph form a tree and the bead string $B_{s,t}$ is a path in this tree, there are no paths $v \sim w$ in $G - e$ for any $w \in B_{s,t}$ because the biconnected components in $G$ are maximal and there would be a larger one (a contradiction). Moreover, let $B_1, B_2, \ldots, B_r$ be the biconnected components composing $B_{s,t}$, where $s \in B_1$ and $t \in B_r$. If there is only one biconnected component in the path (i.e. $r = 1$), there are no articulation points in $B_{s,t}$. Otherwise, all of the $r - 1$ articulation points in $B_{s,t}$ are traversed by each path $\pi \in P_{s,t}(G)$: indeed, the articulation point between adjacent biconnected components $B_i$ and $B_{i+1}$ is their only vertex in common and there are no edges linking $B_i$ and $B_{i+1}. \square
5.2. OVERVIEW AND MAIN IDEAS

We thus restrict the problem of listing the paths in \( P_{s,t}(G) \) to the induced subgraph \( G[B_{s,t}] \), conceptually isolating it from the rest of \( G \). For the sake of description, we will use interchangeably \( B_{s,t} \) and \( G[B_{s,t}] \) in the rest of the chapter.

5.2.3 Binary partition scheme

We list the set of st-paths in \( B_{s,t} \), denoted by \( P_{s,t}(B_{s,t}) \), by applying the binary partition method (where \( P_{s,t}(G) = P_{s,t}(B_{s,t}) \) by Lemma 5.2): we choose an edge \( e = (s, v) \) incident to \( s \) and then list all the st-paths that include \( e \) and then all the st-paths that do not include \( e \). Since we delete some vertices and some edges during the recursive calls, we proceed as follows.

**Invariant:** At a generic recursive step on vertex \( u \) (initially, \( u := s \)), let \( \pi_s = s \leadsto u \) be the path discovered so far (initially, \( \pi_s \) is empty \{\}). Let \( B_{u,t} \) be the current bead string (initially, \( B_{u,t} := B_{s,t} \)). More precisely, \( B_{u,t} \) is defined as follows: (i) remove from \( B_{s,t} \) all the vertices in \( \pi_s \) but \( u \), and the edges incident to \( u \) and discarded so far; (ii) recompute the block tree on the resulting graph; (iii) \( B_{u,t} \) is the unique bead string that connects \( u \) to \( t \) in the recomputed block tree.

**Base case:** When \( u = t \), output the st-path \( \pi_s \).

**Recursive rule:** Let \( P(\pi_s, u, B_{u,t}) \) denote the set of st-paths to be listed by the current recursive call. Then, it is the union of the following two disjoint sets, for an edge \( e = (u, v) \) incident to \( u \):

- **Left branching:** the st-paths in \( P(\pi_s \cdot e, v, B_{v,t}) \) that use \( e \), where \( B_{v,t} \) is the unique bead string connecting \( v \) to \( t \) in the block tree resulting from the deletion of vertex \( u \) from \( B_{u,t} \).

- **Right branching:** the st-paths in \( P(\pi_s, u, B'_{u,t}) \) that do not use \( e \), where \( B'_{u,t} \) is the unique bead string connecting \( u \) to \( t \) in the block tree resulting from the deletion of edge \( e \) from \( B_{u,t} \).

Hence, \( P_{s,t}(B_{s,t}) \) (and so \( P_{s,t}(G) \)) can be computed by invoking \( P(\{\}, s, B_{s,t}) \). The correctness and completeness of the above approach is discussed in Section 5.2.4.

At this point, it should be clear why we introduce the notion of bead strings in the binary partition. The existence of the partial path \( \pi_s \) and the bead string \( B_{u,t} \) guarantees that there surely exists at least one st-path. But there are two sides of the coin when using \( B_{u,t} \).

1. One advantage is that we can avoid useless recursive calls: If vertex \( u \) has only one incident edge \( e \), we just perform the left branching; otherwise, we can safely perform both the left and right branching since the first bead in \( B_{u,t} \) is always a biconnected component by definition (thus there exists both an st-path that traverses \( e \) and one that does not).
2. The other side of the coin is that we have to maintain the bead string $B_{u,t}$ as $B_{v,t}$ in the left branching and as $B'_{u,t}$ in the right branching by Lemma 5.2. Note that these bead strings are surely non-empty since $B_{u,t}$ is non-empty by induction (we only perform either left or left/right branching when there are solutions by item 1).

To efficiently address point 2, we need to introduce the notion of certificate as described next.

### 5.2.4 Introducing the certificate

Given the bead string $B_{u,t}$, we call the head of $B_{u,t}$, denoted by $H_u$, the first biconnected component in $B_{u,t}$, where $u \in H_u$. Consider a DFS tree of $B_{u,t}$ rooted at $u$ that changes along with $B_{u,t}$, and classify the edges in $B_{u,t}$ as tree edges or back edges (there are no cross edges since the graph is undirected).

To maintain $B_{u,t}$ (and so $H_u$) during the recursive calls, we introduce a certificate $C$ (see Fig. 5.4): It is a suitable data structure that uses the above classification of the edges in $B_{u,t}$, and supports the following operations, required by the binary partition scheme.

- **choose**($C, u$): returns an edge $e = (u, v)$ with $v \in H_u$ such that $\pi_s \cdot (u, v) \cdot u \sim t$ is an st-path such that $u \sim t$ is inside $B_{u,t}$. Note that $e$ always exists since $H_u$ is biconnected. Also, the chosen $v$ is the last one in DFS postorder among the neighbors of $u$: in this way, the (only) tree edge $e$ is returned when there are no back edges leaving from $u$. (As it will be clear in Sections 5.3 and 5.4, this order facilitates the analysis and the implementation of the certificate.)

![Figure 5.4: Example certificate of $B_{u,t}$](image)
Algorithm 7 list_paths_s,t(π_s, u, C)

1: if \( u = t \) then
2: \( \text{output}(\pi_s) \)
3: return
4: end if
5: \( e = (u, v) := \text{choose}(C, u) \)
6: if \( e \) is back edge then
7: \( I := \text{right_update}(C, e) \)
8: list_paths_s,t(π_s, u, C)
9: \( \text{restore}(C, I) \)
10: end if
11: \( I := \text{left_update}(C, e) \)
12: list_paths_s,t(π_s · (u, v), v, C)
13: \( \text{restore}(C, I) \)

- \( \text{left_update}(C, e) \): for the given \( e = (u, v) \), it obtains \( B_{v,t} \) from \( B_{u,t} \) as discussed in Section 5.2.3. This implies updating also \( H_u, C, \) and the block tree, since the recursion continues on \( v \). It returns bookkeeping information \( I \) for what is updated, so that it is possible to revert to \( B_{u,t}, H_u, C, \) and the block tree, to their status before this operation.

- \( \text{right_update}(C, e) \): for the given \( e = (u, v) \), it obtains \( B'_{u,t} \) from \( B_{u,t} \) as discussed in Section 5.2.3, which implies updating also \( H_u, C, \) and the block tree. It returns bookkeeping information \( I \) as in the case of \( \text{left_update}(C, e) \).

- \( \text{restore}(C, I) \): reverts the bead string to \( B_{u,t} \), the head \( H_u \), the certificate \( C \), and the block tree, to their status before operation \( I := \text{left_update}(C, e) \) or \( I := \text{right_update}(C, e) \) was issued (in the same recursive call).

Note that a notion of certificate in listing problems has been introduced in [20], but it cannot be directly applied to our case due to the different nature of the problems and our use of more complex structures such as biconnected components.

Using our certificate and its operations, we can now formalize the binary partition and its recursive calls \( \mathcal{P}(\pi_s, u, B_{u,t}) \) described in Section 5.2.3 as Algorithm 7 where \( B_{u,t} \) is replaced by its certificate \( C \).

The base case \( (u = t) \) corresponds to lines 1–4 of Algorithm 7. During recursion, the left branching corresponds to lines 5 and 11-13, while the right branching to lines 5–10. Note that we perform only the left branching when there is only one incident edge in \( u \), which is a tree edge by definition of choose. Also, lines 9 and 13 are needed to restore the parameters to their values when returning from the recursive calls.

**Lemma 5.3.** Algorithm 7 correctly lists all the \( st \)-paths in \( \mathcal{P}_{s,t}(G) \).
CHAPTER 5. LISTING CYCLES AND ST-PATHS

Proof. For a given vertex \( u \) the function \( \text{choose}(C, u) \) returns an edge \( e \) incident to \( u \). We maintain the invariant that \( \pi_s \) is a path \( s \sim u \), since at the point of the recursive call in line 12 (i) is connected as we append edge \((u, v)\) to \( \pi_s \) and; (ii) it is simple as vertex \( u \) is removed from the graph \( G \) in the call to \( \text{left_update}(C, e) \) in line 11. In the case of recursive call in line 8 the invariant is trivially maintained as \( \pi_s \) does not change. The algorithm only outputs \( st \)-paths since \( \pi_s \) is a \( s \sim u \) path and \( u = t \) when the algorithm outputs, in line 2.

The paths with prefix \( \pi_s \) that do not use \( e \) are listed by the recursive call in line 8. This is done by removing \( e \) from the graph (line 7) and thus no path can include \( e \). Paths that use \( e \) are listed in line 12 since in the recursive call \( e \) is added to \( \pi_s \). Given that the tree edge incident to \( u \) is the last one to be returned by \( \text{choose}(C, u) \), there is no path that does not use this edge, therefore it is not necessary to call line 8 for this edge.

A natural question is what is the time complexity: we must account for the cost of maintaining \( C \) and for the cost of the recursive calls of Algorithm 7. Since we cannot always maintain the certificate in \( O(1) \) time, the ideal situation for attaining an optimal cost is taking \( O(\mu) \) time if at least \( \mu \) \( st \)-paths are listed in the current call (and its nested calls). Unfortunately, we cannot estimate \( \mu \) efficiently and cannot design Algorithm 7 so that it takes \( O(\mu) \) adaptively. We circumvent this by using a different cost scheme in Section 5.2.5 that is based on the recursion tree induced by Algorithm 7. Section 5.4 is devoted to the efficient implementation of the above certificate operations according to the cost scheme that we discuss next.

5.2.5 Recursion tree and cost amortization

We now show how to distribute the costs among the several recursive calls of Algorithm 7 so that optimality is achieved. Consider a generic execution on the bead string \( B_{u,t} \). We trace this execution by using a binary recursion tree \( R \). The nodes of \( R \) are labeled by the arguments of Algorithm 7: specifically, we denote a node in \( R \) by the triple \( x = (\pi_s, u, C) \) iff it represents the call with arguments \( \pi_s, u, \) and \( C \). The left branching is represented by the left child, and the right branching (if any) by the right child of the current node.

Lemma 5.4. The binary recursion tree \( R \) for \( B_{u,t} \) has the following properties:

1. There is a one-to-one correspondence between the paths in \( P_{s,t}(B_{u,t}) \) and the leaves in the recursion tree rooted at node \( (\pi_s, u, C) \).

2. Consider any leaf and its corresponding \( st \)-path \( \pi \): there are \( |\pi| \) left branches in the corresponding root-to-leaf trace.

\(^1\)For clarity, we use “nodes” when referring to \( R \) and “vertices” when referring to \( B_{u,t} \).
5.2. OVERVIEW AND MAIN IDEAS

3. Consider the instruction \( e := \text{choose}(C, u) \) in Algorithm \( \text{[?]}. \) Unary (i.e. single-child) nodes correspond to left branches (\( e \) is a tree edge) while binary nodes correspond to left and right branches (\( e \) is a back edge).

4. The number of binary nodes is \( |P_{st}(B_{u,t})| - 1 \).

Proof. We proceed in order as follows.

1. We only output a solution in a leaf and we only do recursive calls that lead us to a solution. Moreover every node partitions the set of solutions in the ones that use an edge and the ones that do not use it. This guarantees that the leaves in the left subtree of the node corresponding to the recursive call and the leaves in the right subtree do not intersect. This implies that different leaves correspond to different paths from \( s \) to \( t \), and that for each path there is a corresponding leaf.

2. Each left branch corresponds to the inclusion of an edge in the path \( \pi \).

3. Since we are in a biconnected component, there is always a left branch. There can be no unary node as a right branch: indeed for any edge of \( B_{u,t} \) there exists always a path from \( s \) to \( t \) passing through that edge. Since the tree edge is always the last one to be chosen, unary nodes cannot correspond to back edges and binary nodes are always back edges.

4. It follows from point \( [1] \) and from the fact that the recursion tree is a binary tree. (In any binary tree, the number of binary nodes is equal to the number of leaves minus 1.)

We define a spine of \( R \) to be a subset of \( R \)'s nodes linked as follows: the first node is a node \( x \) that is either the left child of its parent or the root of \( R \), and the other nodes are those reachable from \( x \) by right branching in \( R \). Let \( x = (\pi_x, u, C) \) be the first node in a spine \( S \). The nodes in \( S \) correspond to the edges that are incident to vertex \( u \) in \( B_{u,t} \): hence their number equals the degree \( d(u) \) of \( u \) in \( B_{u,t} \),
and the deepest (last) node in $S$ is always a tree edge in $B_{u,t}$ while the others are back edges. Fig. 5.5 shows the spine corresponding to $B_{u,t}$ in Fig. 5.4. Summing up, $R$ can be seen as composed by spines, unary nodes, and leaves where each spine has a unary node as deepest node. This gives a global picture of $R$ that we now exploit for the analysis.

We define the compact head, denoted by $H_X = (V_X, E_X)$, as the (multi)graph obtained by compacting the maximal chains of degree-2 vertices, except $u$, $t$, and the vertices that are the leaves of its DFS tree rooted at $u$.

The rationale behind the above definition is that the costs defined in terms of $H_X$ amortize well, as the size of $H_X$ and the number of st-paths in the subtree of $R$ rooted at node $x = \langle \pi_s, u, C \rangle$ are intimately related (see Lemma 5.6 in Section 5.3) while this is not necessarily true for $H_u$.

Recall that each leaf corresponds to a path $\pi$ and each spine corresponds to a compact head $H_X = (V_X, E_X)$. We now define the following abstract cost for spines, unary nodes, and leaves of $R$, for a sufficiently large constant $c_0 > 0$, that Algorithm 7 must fulfill:

$$
T(r) = \begin{cases} 
  c_0 & \text{if } r \text{ is unary} \\
  c_0|\pi| & \text{if } r \text{ is a leaf} \\
  c_0(|V_X| + |E_X|) & \text{if } r \text{ is a spine}
\end{cases} 
$$

(5.1)

**Lemma 5.5.** The sum of the costs in the nodes of the recursion tree $\sum_{r \in R} T(r) = O(\sum_{\pi \in P_{s,t}(B_{u,t})} |\pi|)$.

Section 5.3 contains the proof of Lemma 5.5 and related properties. Setting $u := s$, we obtain that the cost in Lemma 5.5 is optimal, by Lemma 5.2.

**Theorem 5.3.** Algorithm 7 solves problem Problem 5.1 in $O(m + \sum_{\pi \in P_{s,t}(G)} |\pi|)$ time.

By Lemma 5.1, we obtain an optimal result for listing cycles.

**Theorem 5.4.** Problem 5.2 can be optimally solved in $O(m + \sum_{c \in C(G)} |c|)$ time.

### 5.3 Amortization strategy

We devote this section to prove Lemma 5.5. Let us split the sum in Eq. (5.1) in three parts, and bound each part individually, as

$$
\sum_{r \in R} T(r) \leq \sum_{r: \text{unary}} T(r) + \sum_{r: \text{leaf}} T(r) + \sum_{r: \text{spine}} T(r). 
$$

(5.2)

We have that $\sum_{r: \text{unary}} T(r) = O(\sum_{\pi \in P_{s,t}(G)} |\pi|)$, since there are $|P_{s,t}(G)|$ leaves, and the root-to-leaf trace leading to the leaf for $\pi$ contains at most $|\pi|$ unary nodes by Lemma 5.4, where each unary node has cost $O(1)$ by Eq. (5.1).
Also, \( \sum_{r: \text{leaf}} T(r) = O(\sum_{\pi \in \mathcal{P}_{\pi}(G)} |\pi|) \), since the leaf \( r \) for \( \pi \) has cost \( O(|\pi|) \) by Eq. (5.1).

It remains to bound \( \sum_{r: \text{spine}} T(r) \). By Eq. (5.1), we can rewrite this cost as \( \sum_{H_X} c_0(|V_X| + |E_X|) \), where the sum ranges over the compacted heads \( H_X \) associated with the spines \( r \). We use the following lemma to provide a lower bound on the number of \( st \)-paths descending from \( r \).

**Lemma 5.6.** Given a spine \( r \), and its head string \( B_{u,t} \) with head \( H_u \), there are at least \( |E_X| - |V_X| + 1 \) \( st \)-paths in \( G \) that have prefix \( \pi_s = s \leadsto u \) and suffix \( u \leadsto t \) internal to \( B_{u,t} \), where the compacted head is \( H_X = (V_X, E_X) \).

**Proof.** \( H_X \) is biconnected. In any biconnected graph \( B = (V_B, E_B) \) there are at least \( |E_B| - |V_B| + 1 \) \( xy \)-paths for any \( x, y \in V_B \). Find an ear decomposition \([16]\) of \( B \) and consider the process of forming \( B \) by adding ears one at the time, starting from a single cycle including \( x \) and \( y \). Initially \( |V_B| = |E_B| \) and there are 2 \( xy \)-paths. Each new ear forms a path connecting two vertices that are part of a \( xy \)-path, increasing the number of paths by at least 1. If the ear has \( k \) edges, its addition increases \( V \) by \( k - 1 \), \( E \) by \( k \), and the number of \( xy \)-paths by at least 1. The result follows by induction.

The implication of Lemma 5.6 is that there are at least \( |E_X| - |V_X| + 1 \) leaves descending from the given spine \( r \). Hence, we can charge to each of them a cost of \( c_0(|V_X| + |E_X|) \). Lemma 5.7 allows us to prove that the latter cost is \( O(1) \) when \( H_u \) is different from a single edge or a cycle. (If \( H_u \) is a single edge or a cycle, \( H_X \) is a single or double edge, and the cost is trivially a constant.)

**Lemma 5.7.** For a compacted head \( H_X = (V_X, E_X) \), its density is \( \frac{|E_X|}{|V_X|} \geq \frac{11}{10} \).

**Proof.** Consider the following partition \( V_X = \{r\} \cup V_2 \cup V_3 \) where: \( r \) is the root; \( V_2 \) is the set of vertices with degree 2 and; \( V_3 \), the vertices with degree \( \geq 3 \). Since \( H_X \) is compacted DFS tree of a biconnected graph, we have that \( V_2 \) is a subset of the leaves and \( V_3 \) contains the set of internal vertices (except \( r \)). There are no vertices with degree 1 and \( d(r) \) \( \geq 2 \). Let \( x = \sum_{v \in V_3} d(v) \) and \( y = \sum_{v \in V_2} d(v) \). We can write the density as a function of \( x \) and \( y \), namely,

\[
\frac{|E_X|}{|V_X|} = \frac{x + y + d(r)}{2(|V_3| + |V_2| + 1)}
\]

Note that \( |V_3| \leq \frac{x}{4} \) as the vertices in \( V_3 \) have at least degree 3, \( |V_2| = \frac{x}{2} \) as vertices in \( V_2 \) have degree exactly 2. Since \( d(r) \geq 2 \), we derive the following bound

\[
\frac{|E_X|}{|V_X|} \geq \frac{x + y + 2}{\frac{x}{2} + y + 2}
\]

Consider any graph with \( |V_X| > 3 \) and its DFS tree rooted at \( r \). Note that: (i) there are no tree edges between any two leaves, (ii) every vertex in \( V_2 \) is a leaf and
(iii) no leaf is a child of \( r \). Therefore, every tree edge incident in a vertex of \( V_2 \) is also incident in a vertex of \( V_3 \). Since exactly half the incident edges to \( V_2 \) are tree edges (the other half are back edges) we get that \( y \leq 2x \).

With \( |V_X| \geq 3 \) there exists at least one internal vertex in the DFS tree and therefore \( x \geq 3 \).

\[
\text{minimize } \quad \frac{x + y + 2}{\frac{2}{3}x + y + 2} \\
\text{subject to } \quad 0 \leq y \leq 2x, \\
\quad x \geq 3.
\]

Since for any \( x \) the function is minimized by the maximum \( y \) s.t. \( y \leq 2x \) and for any \( y \) by the minimum \( x \), we get

\[
\frac{|E_X|}{|V_X|} \geq \frac{9x + 6}{8x + 6} \geq \frac{11}{10}.
\]

Specifically, let \( \alpha = \frac{11}{10} \) and write \( \alpha = 1 + 2/\beta \) for a constant \( \beta \): we have that

\[
|E_X| + |V_X| = (|E_X| - |V_X|) + 2|V_X| \leq (|E_X| - |V_X|) + \beta(|E_X| - |V_X|) = \frac{\alpha + 1}{\alpha - 1}(|E_X| - |V_X|)
\]

Thus, we can charge each leaf with a cost of \( c_0\frac{\alpha + 1}{\alpha - 1} \leq c_0 = O(1) \). This motivates the definition of \( H_X \), since Lemma 5.7 does not necessarily hold for the head \( H_u \) (due to the unary nodes in its DFS tree).

One last step to bound \( \sum H_X c_0(|V_X| + |E_X|) \): as noted before, a root-to-leaf trace for the string storing \( \pi \) has \( |\pi| \) left branches by Lemma 5.4, and as many spines, each spine charging \( c_0\frac{\alpha + 1}{\alpha - 1} = O(1) \) to the leaf at hand. This means that each of the \( |P_{s,t}(G)| \) leaves is charged for a cost of \( O(|\pi|) \), thus bounding the sum as

\[
\sum_{r \text{ spine}} T(r) = \sum H_X c_0(|V_X| + |E_X|) = O(\sum_{\pi \in P_{s,t}(G)} |\pi|)
\]

This completes the proof of Lemma 5.5. As a corollary, we obtain the following result.

**Lemma 5.8.** The recursion tree \( R \) with cost as in Eq. (5.1) induces an \( O(|\pi|) \) amortized cost for each \( st \)-path \( \pi \).

### 5.4 Certificate implementation and maintenance

We show how to represent and update the certificate \( C \) so that the time taken by Algorithm 7 in the recursion tree can be distributed among the nodes as in Eq. (5.1): namely, (i) \( O(1) \) time in unary nodes; (ii) \( O(|\pi|) \) time in the leaf corresponding to path \( \pi \); (iii) \( O(|V_X| + |E_X|) \) in each spine.

The certificate \( C \) associated with a node \( \langle \pi_s, u, C \rangle \) in the recursion tree is a compacted and augmented DFS tree of bead string \( B_{u,t} \), rooted at vertex \( u \). The DFS tree changes over time along with \( B_{u,t} \), and is maintained in such a way that \( t \) is in the leftmost path of the tree.
We augment the DFS tree with sufficient information to implicitly represent articulation points and biconnected components. Recall that, by the properties of a DFS tree of an undirected graph, there are no cross edges (edges between different branches of the tree).

We compact the DFS tree by contracting the vertices that have degree 2, except $u$, $t$, and the leaves (the latter surely have incident back edges). Maintaining this compacted representation is not a difficult data-structure problem. From now on we can assume w.l.o.g. that $C$ is an augmented DFS tree rooted at $u$ where internal nodes of the DFS tree have degree $\geq 3$, and each vertex $v$ has associated the following information.

1. A doubly-linked list $lb(v)$ of back edges linking $v$ to its descendants $w$ sorted by postorder DFS numbering.
2. A doubly-linked list $ab(v)$ of back edges linking $v$ to its ancestors $w$ sorted by preorder DFS numbering.
3. An integer $\gamma(v)$, such that if $v$ is an ancestor of $w$ then $\gamma(v) < \gamma(w)$.
4. The smallest $\gamma(w)$ over all $w$, such that $(h,w)$ is a back edge and $h$ is in the subtree of $v$, denoted by $\text{lowpoint}(v)$.

Given three vertices $v,w,x \in C$ such that $v$ is the parent of $w$ and $x$ is not in the subtree of $w$, we can efficiently test if $v$ is an articulation point, i.e. $\text{lowpoint}(w) \leq \gamma(v)$. (Note that we adopt a variant of $\text{lowpoint}$ using $\gamma(v)$ in place of the preorder numbering \cite{SS}; it has the same effect whereas using $\gamma(v)$ is preferable since it is easier to dynamically maintain.)

**Lemma 5.9.** The certificate associated with the root of the recursion can be computed in $O(m)$ time.

**Proof.** In order to set $t$ to be in the leftmost path, we perform a DFS traversal of graph $G$ starting from $s$ and stop when we reach vertex $t$. We then compute the DFS tree, traversing the path $s \leadsto t$ first. When visiting vertex $v$, we set $\gamma(v)$ to depth of $v$ in the DFS. Before going up on the traversal, we compute the lowpoints using the lowpoints of the children. Let $z$ be the parent of $v$. If $\text{lowpoint}(v) \leq \gamma(z)$ and $v$ is not in the leftmost path in the DFS, we cut the subtree of $v$ as it does not belong to $B_{s,t}$. When first exploring the neighborhood of $v$, if $w$ was already visited, i.e. $e = (u,w)$ is a back edge, and $w$ is a descendant of $v$; we add $e$ to $ab(w)$. This maintains the DFS preordering in the ancestor back edge list. Now, after the first scan of $N(v)$ is over and all the recursive calls returned (all the children were explored), we re-scan the neighborhood of $v$. If $e = (v,w)$ is a back edge and $w$ is an ancestor of $v$, we add $e$ to $lb(w)$. This maintains the DFS postorder in the descendant back edge list. This procedure takes at most two DFS traversals in $O(m)$ time. This DFS tree can be compacted in the same time bound. \hfill $\square$

\footnote{The second condition is always satisfied when $w$ is not in the leftmost path, since $t$ is not in the subtree of $w$.}
Lemma 5.10. Operation \textit{choose}(C, u) can be implemented in \textit{O}(1) time.

\textit{Proof.} If the list \( lb(v) \) is empty, return the tree edge \( e = (u, v) \) linking \( u \) to its only child \( v \) (there are no other children). Else, return the last edge in \( lb(v) \). \hfill \square

We analyze the cost of updating and restoring the certificate \( C \). We can reuse parts of \( C \), namely, those corresponding to the vertices that are not in the compacted head \( H_X = (V_X, E_X) \) as defined in Section 5.2.5. We prove that, given a unary node \( u \) and its tree edge \( e = (u, v) \), the subtree of \( v \) in \( C \) can be easily made a certificate for the left branch of the recursion.

Lemma 5.11. On a unary node, \textit{left_update}(C, e) takes \textit{O}(1) time.

\textit{Proof.} Take edge \( e = (u, v) \). Remove edge \( e \) and set \( v \) as the root of the certificate. Since \( e \) is the only edge incident in \( v \), the subtree \( v \) is still a DFS tree. Cut the list of children of \( v \) keeping only the first child. (The other children are no longer in the bead string and become part of \( I \).) There is no need to update \( \gamma(v) \). \hfill \square

We now devote the rest of this section to show how to efficiently maintain \( C \) on a spine. Consider removing a back edge \( e \) from \( u \): the compacted head \( H_X = (V_X, E_X) \) of the bead string can be divided into smaller biconnected components. Many of those can be excluded from the certificate (i.e. they are no longer in the new bead string, and so they are bookkept in \( I \)) and additionally we have to update the lowpoints that change. We prove that this operation can be performed in \textit{O}(|V_X|) total time on a spine of the recursion tree.

Lemma 5.12. The total cost of all the operations \textit{right_update}(C, e) in a spine is \textit{O}(|V_X|) time.

\textit{Proof.} In the right branches along a spine, we remove all back edges in \( lb(u) \). This is done by starting from the last edge in \( lb(u) \), i.e. proceeding in reverse DFS postorder. For back edge \( b_i = (z_i, u) \), we traverse the vertices in the path from \( z_i \) towards the root \( u \), as these are the only lowpoints that can change. While moving upwards on the tree, on each vertex \( w \), we update \textit{lowpoint}(w). This is done by taking the endpoint \( y \) of the first edge in \( ab(w) \) (the back edge that goes the topmost in the tree) and choosing the minimum between \( \gamma(y) \) and the lowpoint of each child of \( w \). We stop when the updated \textit{lowpoint}(w) = \( \gamma(u) \) since it implies that the lowpoint of the vertex can not be further reduced. Note that we stop before \( u \), except when removing the last back edge in \( lb(u) \).

To prune the branches of the DFS tree that are no longer in \( B_{u,t} \), consider again each vertex \( w \) in the path from \( z_i \) towards the root \( u \) and its parent \( y \). We check

\[ \text{If} \ \text{lowpoint}(w) \ \text{does not change we cannot pay to explore its children. For each vertex we dynamically maintain a list } l(w) \ \text{of its children that have lowpoint equal to } \gamma(u). \ \text{Then, we can test in constant time if } l(w) \neq \emptyset \ \text{and } y \text{ is not the root } u. \ \text{If both conditions are true lowpoint}(w) \text{ changes, otherwise it remains equal to } \gamma(u) \ \text{and we stop.} \]
if the updated $\text{lowpoint}(w) \leq \gamma(y)$ and $w$ is not in the leftmost path of the DFS. If both conditions are satisfied, we have that $w \notin B_{u,t}$, and therefore we cut the subtree of $w$ and keep it in $I$ to restore later. We use the same halting criterion as in the previous paragraph.

The cost of removing all back edges in the spine is $O(|V_X|)$: there are $O(|V_X|)$ tree edges and, in the paths from $z_i$ to $u$, we do not traverse the same tree edge twice since the process described stops at the first common ancestor of endpoints of back edges $b_i$. Additionally, we take $O(1)$ time to cut a subtree of an articulation point in the DFS tree.

To compute $\text{left} \ update(C,e)$ in the binary nodes of a spine, we use the fact that in every left branching from that spine, the graph is the same (in a spine we only remove edges incident to $u$ and on a left branch from the spine we remove the vertex $u$) and therefore its block tree is also the same. However, the certificates on these nodes are not the same, as they are rooted at different vertices. Using the reverse DFS postorder of the edges, we are able to traverse each edge in $H_X$ only a constant number of times in the spine.

Lemma 5.13. The total cost of all operations $\text{left} \ update(C,e)$ in a spine is amortized $O(|E_X|)$.

Proof. Let $t'$ be the last vertex in the path $u \leadsto t$ s.t. $t' \in V_X$. Since $t'$ is an articulation point, the subtree of the DFS tree rooted in $t'$ is maintained in the case of removal of vertex $u$. Therefore the only modifications of the DFS tree occur in the compacted head $H_X$ of $B_{u,t}$. Let us compute the certificate $C_i$: this is the certificate of the left branch of the $i$th node of the spine where we augment the path with the back edge $b_i = (z_i, u)$ of $lb(u)$ in the order defined by $\text{choose}(C,u)$.

For the case of $C_1$, we remove $u$ and rebuild the certificate starting form $z_1$ (the last edge in $lb(u)$) using the algorithm from Lemma 5.9 restricted to $H_X$ and using $t'$ as target and $\gamma(t')$ as a baseline to $\gamma$ (instead of the depth). This takes $O(|E_X|)$ time.

For the general case of $C_i$ with $i > 1$ we also rebuild (part) of the certificate starting from $z_i$ using the procedure from Lemma 5.9 but we use information gathered in $C_{i-1}$ to avoid exploring useless branches of the DFS tree. The key point is that, when we reach the first bead in common to both $B_{z_i,t}$ and $B_{z_{i-1},t}$, we only explore edges internal to this bead. If an edge $e$ leaving the bead leads to $t$, we can reuse a subtree of $C_{i-1}$. If $e$ does not lead to $t$, then it has already been explored (and cut) in $C_{i-1}$ and there is no need to explore it again since it will be discarded. Given the order we take $b_i$, each bead is not added more than once, and the total cost over the spine is $O(|E_X|)$.

Nevertheless, the internal edges $E'_X$ of the first bead in common between $B_{z_i,t}$ and $B_{z_{i-1},t}$ can be explored several times during this procedure.\footnote{Consider the case where $z_i, \ldots, z_j$ are all in the same bead after the removal of $u$. The bead}
cost $O(|E'_X|)$ of exploring those edges to another node in the recursion tree, since this common bead is the head of at least one certificate in the recursion subtree of the left child of the $i$th node of the spine. Specifically, we charge the first node in the leftmost path of the $i$th node of the spine that has exactly the edges $E'_X$ as head of its bead string: (i) if $|E'_X| \leq 1$ it corresponds to a unary node or a leaf in the recursion tree and therefore we can charge it with $O(1)$ cost; (ii) otherwise it corresponds to a first node of a spine and therefore we can also charge it with $O(|E'_X|)$. We use this charging scheme when $i \neq 1$ and the cost is always charged in the leftmost recursion path of $i$th node of the spine. Consequently, we never charge a node in the recursion tree more than once.

**Lemma 5.14.** On each node of the recursion tree, $\text{restore}(C, I)$ takes time proportional to the size of the modifications kept in $I$.

**Proof.** We use standard data structures (i.e. linked lists) for the representation of certificate $C$. Persistent versions of these data structures exist that maintain a stack of modifications applied to them and that can restore its contents to their previous states. Given the modifications in $I$, these data structures take $O(|I|)$ time to restore the previous version of $C$.

Let us consider the case of performing $\text{left_update}(C, e)$. We cut at most $O(|V_X|)$ edges from $C$. Note that, although we conceptually remove whole branches of the DFS tree, we only remove edges that attach those branches to the DFS tree. The other vertices and edges are left in the certificate but, as they no longer remain attached to $B_{u,t}$, they will never be reached or explored. In the case of $\text{right_update}(C, e)$, we have a similar situation, with at most $O(|E_X|)$ edges being modified along the spine of the recursion tree.

From Lemmas 5.10 and 5.12, it follows that on a spine of the recursion tree we have the costs: $\text{choose}(u)$ on each node which is bounded by $O(|V_X|)$ time as there are at most $|V_X|$ back edges in $u$; $\text{right_update}(C, e), \text{restore}(C, I)$ take $O(|V_X|)$ time; $\text{left_update}(C, e)$ and $\text{restore}(C, I)$ are charged $O(|V_X| + |E_X|)$ time. We thus have the following result, completing the proof of Theorem 5.3.

**Lemma 5.15.** Algorithm 7 can be implemented with a cost fulfilling Eq. (5.1), thus it takes total $O(m + \sum_{r \in R} T(r)) = O(m + \sum_{\pi \in \mathcal{P}_{s,t}(B_{u,t})} |\pi|)$ time.

### 5.5 Extended analysis of operations

In this suplementary section, we present all details and illustrate with figures the operations $\text{right_update}(C, e)$ and $\text{left_update}(C, e)$ that are performed along a spine of the recursion tree. In order to better detail the procedures in Lemma 5.12, strings are the same, but the roots $z_i, \ldots, z_j$ are different, so we have to compute the corresponding DFS of the first component $|j - i|$ times.
5.5. EXTENDED ANALYSIS OF OPERATIONS

We use bead string $B_{u,t}$ from Fig. 5.4 and the respective spine from Fig. 5.5 as the base for the examples. This spine contains four binary nodes corresponding to the back edges in $lb(u)$ and an unary node corresponding to the tree edge $(u,v)$. Note that edges are taken in order of the endpoints $z_1, z_2, z_3, z_4, v$ as defined in operation $\text{choose}(C,u)$.

By Lemma 5.2, the impact of operations $\text{right}_\text{update}(C,e)$ and $\text{left}_\text{update}(C,e)$ in the certificate is restricted to the biconnected component of $u$. Thus we mainly focus on maintaining the compacted head $H_X = (V_X, E_X)$ of the bead string $B_{u,t}$.

5.5.1 Operation $\text{right}_\text{update}(C,e)$

Lemma 5.16. (Lemma 5.12 restated) In a spine of the recursion tree, operations $\text{right}_\text{update}(C,e)$ can be implemented in $O(|V_X|)$ total time.

In the right branches along a spine, we remove all back edges in $lb(u)$. This is done by starting from the last edge in $lb(u)$, i.e. proceeding in reverse DFS postorder. In the example from Fig. 5.4, we remove the back edges $(z_1, u)$, $v$ as defined in operation $\text{choose}(C,u)$.

Consider the update of lowpoints in the DFS tree. For a back edge $b_i = (z_i, u)$, we traverse the vertices in the path from $z_i$ towards the root $u$. By definition of lowpoint, these are the only lowpoints that can change. Suppose that we remove back edge $(z_4, u)$ in the example from Fig. 5.4. Only the lowpoints of the vertices in the path from $z_4$ towards the root $u$ change. Furthermore, consider a vertex $w$ in the tree that is an ancestor of at least two endpoints $z_i, z_j$ of back edges $b_i, b_j$. The lowpoint of $w$ does not change when we remove $b_i$. These observations lead us to the following lemma.

Figure 5.6: Application of $\text{right}_\text{update}(C, e)$ on a spine of the recursion tree
Lemma 5.17. In a spine of the recursion tree, the update of lowpoints in the certificate by operation right_update($C,e$) can be done in $O(|V_X|)$ total time.

Proof. Take each back edge $b_i = (z_i,u)$ in the order defined by choose($C,u$). Remove $b_i$ from $lb(u)$ and $ab(z_i)$. Starting from $z_i$, consider each vertex $w$ in the path from $z_i$ towards the root $u$. On vertex $w$, we update lowpoint($w$) using the standard procedure: take the endpoint $y$ of the first edge in $ab(w)$ (the back edge that goes the nearest to the root of the tree) and choosing the minimum between $\gamma(y)$ and the lowpoint of each child of $w$. When the updated lowpoint($w$) = $\gamma(u)$, we stop examining the path from $z_i$ to $u$ since it implies that the lowpoint of the vertex can not be further reduced (i.e. $w$ is both an ancestor to both $z_i$ and $z_{i+1}$).

If lowpoint($w$) does not change we cannot pay to explore its children. In order to get around this, for each vertex we dynamically maintain, throughout the spine, a list $l(w)$ of its children that have lowpoint equal to $\gamma(u)$. Then, we can test in constant time if $l(w) \neq \emptyset$ and $y$ (the endpoint of the first edge in $ab(w)$) is not the root $u$. If both conditions are satisfied lowpoint($w$) changes, otherwise it remains equal to $\gamma(u)$ and we stop. The total time to create the lists is $O(|V_X|)$ and the time to update is bounded by the number of tree edges traversed, shown to be $O(|V_X|)$ in the next paragraph.

The cost of updating the lowpoints when removing all back edges $b_i$ is $O(|V_X|)$: there are $O(|V_X|)$ tree edges and we do not traverse the same tree edge twice since the process described stops at the first common ancestor of endpoints of back edges $b_i$ and $b_{i+1}$. By contradiction: if a tree edge $(x,y)$ would be traversed twice when removing back edges $b_i$ and $b_{i+1}$, it would imply that both $x$ and $y$ are ancestors of $z_i$ and $z_{i+1}$ (as edge $(x,y)$ is both in the path $z_i$ to $u$ and the path $z_{i+1}$ to $u$) but we stop at the first ancestor of $z_i$ and $z_{i+1}$.

Let us now consider the removal of vertices that are no longer in $B_{u,t}$ as consequence of operation right_update($C,e$) in a spine of the recursion tree. By removing a back edge $b_i = (z_i,u)$, it is possible that a vertex $w$ previously in $H_X$ is no longer in the bead string $B_{u,t}$ (e.g. $w$ is no longer biconnected to $u$ and thus there is no simple path $u \rightsquigarrow w \rightsquigarrow t$).

Lemma 5.18. In a spine of the recursion tree, the branches of the DFS that are no longer in $B_{u,t}$ due to operation right_update($C,e$) can be removed from the certificate in $O(|V_X|)$ total time.

Proof. To prune the branches of the DFS tree that are no longer in $H_X$, consider again each vertex $w$ in the path from $z_i$ towards the root $u$ and the vertex $y$, parent of $w$. It is easy to check if $y$ is an articulation point by verifying if the updated lowpoint($w$) $\leq \gamma(y)$ and there exists $x$ not in the subtree of $w$. If $w$ is not in the leftmost path, then $t$ is not in the subtree of $w$. If that is the case, we have that $w \notin B_{u,t}$, and therefore we cut the subtree of $w$ and bookkeep it in $I$ to restore later. Like in the update the lowpoints, we stop examining the path $z_i$ towards $u$ in
5.5. EXTENDED ANALYSIS OF OPERATIONS

When cutting the subtree of \( w \), note that there are no back edges connecting it to \( B_{u,t} \) (\( w \) is an articulation point) and therefore there are no updates to the lists \( lb \) and \( ab \) of the vertices in \( B_{u,t} \). Like in the case of updating the lowpoints, we do not traverse the same tree edge twice (we use the same halting criterion).

With Lemma 5.17 and Lemma 5.18 we finalize the proof of Lemma 5.12. Fig. 5.6 shows the changes the bead string \( B_{u,t} \) from Fig. 5.4 goes through in the corresponding spine of the recursion tree.

5.5.2 Operation left_update\((C,e)\)

In the binary nodes of a spine, we use the fact that in every left branching from that spine the graph is the same (in a spine we only remove edges incident to \( u \) and on a left branch from the spine we remove the vertex \( u \)) and therefore its block tree is also the same. In Fig. 5.7 we show the resulting block tree of the graph from Fig. 5.4 after having removed vertex \( u \). However, the certificates on these left branches are not the same, as they are rooted at different vertices. In the example we must compute the certificates \( C_1 \ldots C_4 \) corresponding to bead strings \( B_{z_1,t} \ldots B_{z_4,t} \). We do not account for the cost of the left branch on the last node of spine (corresponding to \( B_{v,t} \)) as the node is unary and we have shown in Lemma 5.11 how to maintain the certificate in \( O(1) \) time.

By using the reverse DFS postorder of the back edges, we are able to traverse each edge in \( H_X \) only an amortized constant number of times in the spine.

Lemma 5.19. (Lemma 5.13 restated) The calls to operation \( \text{left}_\text{update}(C,e) \) in a spine of the recursion tree can be charged with a time cost of \( O(|E_X|) \) to that spine.

To achieve this time cost, for each back edge \( b_i = (z_i,u) \), we compute the certificate corresponding to \( B_{z_1,t} \) based on the certificate of \( B_{z_1-1,t} \). Consider the compacted head \( H_X = (V_X,E_X) \) of the bead string \( B_{u,t} \). We use \( O(|E_X|) \) time to compute the first certificate \( C_1 \) corresponding to bead string \( B_{z_1,t} \). Fig. 5.8 shows bead string \( B_{z_1,t} \) from the example of Fig. 5.4.

Lemma 5.20. The certificate \( C_1 \), corresponding to bead string \( B_{z_1,t} \), can be computed in \( O(|E_X|) \) time.

Proof. Let \( t' \) be the last vertex in the path \( u \leadsto t \) s.t. \( t' \in V_X \). Since \( t' \) is an articulation point, the subtree of the DFS tree rooted in \( t' \) is maintained in the case of removal vertex \( u \). Therefore the only modifications of the DFS tree occur in head \( H_X \) of \( B_{u,t} \).

To compute \( C_1 \), we remove \( u \) and rebuild the certificate starting form \( z_1 \) using the algorithm from Lemma 5.9 restricted to \( H_X \) and using \( t' \) as target and \( \gamma(t') \) as a baseline to \( \gamma \) (instead of the depth). In particular we do the following. To set \( t' \)
to be in the leftmost path, we perform a DFS traversal of graph $H_X$ starting from $z_1$ and stop when we reach vertex $t'$. Then compute the DFS tree, traversing the path $z_1 \leadsto t'$ first.

Update of $\gamma$. For each tree edge $(v, w)$ in the $t' \leadsto z_1$ path, we set $\gamma(v) = \gamma(w) - 1$, using $\gamma(t')$ as a baseline. During the rest of the traversal, when visiting vertex $v$, let $w$ be the parent of $v$ in the DFS tree. We set $\gamma(v) = \gamma(w) + 1$. This maintains the property that $\gamma(v) > \gamma(w)$ for any $w$ ancestor of $v$.

Lowpoints and pruning the tree. Bottom-up in the DFS-tree, compute the lowpoints using the lowpoints of the children. For $z$ the parent of $v$, if $\text{lowpoint}(v) \leq \gamma(z)$ and $v$ is not in the leftmost path in the DFS, cut the subtree of $v$ as it does not belong to $B_{z_1,t}$.

Computing $lb$ and $ab$. In the traversal, when finding a back edge $e = (v, w)$, if $w$ is a descendant of $v$ we append $e$ to $ab(w)$. This maintains the DFS preorder in the ancestor back edge list. After the first scan of $N(v)$ is over and all the recursive calls returned, re-scan the neighborhood of $v$. If $e = (v, w)$ is a back edge and $w$ is an ancestor of $v$, we add $e$ to $lb(w)$. This maintains the DFS postorder in the descendant back edge list. This procedure takes $O(|E_X|)$ time.

To compute each certificate $C_i$, corresponding to bead string $B_{z_i,t}$, we are able to avoid visiting most of the edges that belong to $B_{z_{i-1},t}$. Since we take $z_i$ in reverse DFS postorder, on the spine of the recursion we visit $O(|E_X|)$ edges plus a term

Figure 5.7: Block tree after removing vertex $u$
that can be amortized.

**Lemma 5.21.** For each back edge \( b_i = (z_i, u) \) with \( i > 1 \), let \( E'_{X_i} \) be the edges in the first bead in common between \( B_{z_i,t} \) and \( B_{z_i-1,t} \). The total cost of computing all certificates \( B_{z_i,t} \) in a spine of the recursion tree is: \( O(|E_X| + \sum_{i>1} |E'_{X_i}|) \).

**Proof.** Let us compute the certificate \( C_i \): the certificate of the left branch of the \( i \)-th node of the spine where we augment the path with back edge \( b_i = (z_i, u) \) of \( lb(u) \).

For the general case of \( C_i \) with \( i > 1 \) we also rebuild (part) of the certificate starting from \( z_i \) using the procedure from Lemma 5.9 but we use information gathered in \( C_{i-1} \) to avoid exploring useless branches of the DFS tree. The key point is that, when we reach the first bead in common to both \( B_{z_i,t} \) and \( B_{z_i-1,t} \), we only explore edges internal to this bead. If an edge \( e \) that leaves the bead leads to \( t \), we can reuse a subtree of \( C_{i-1} \). If \( e \) does not lead to \( t \), then it has already been explored (and cut) in \( C_{i-1} \) and there is no need to explore it again since it is going to be discarded.

In detail, we start computing a DFS from \( z_i \) in \( B_{z_i,t} \) until we reach a vertex \( t' \in B_{z_i-1,t} \). Note that the bead of \( t' \) has one entry point and one exit point in \( C_{i-1} \). After reaching \( t' \) we proceed with the traversal using only edges already in \( C_{i-1} \). When arriving at a vertex \( w \) that is not in the same bead of \( t' \), we stop the traversal. If \( w \) is in a bead towards \( t \), we reuse the subtree of \( w \) and use \( \gamma(w) \) as a baseline of the numbering \( \gamma \). Otherwise \( w \) is in a bead towards \( z_i-1 \) and we cut this branch of the certificate. When all edges in the bead of \( t' \) are traversed, we proceed with visit in the standard way.

Given the order we take \( b_i \), each back edge is not added more than once to a certificate \( C_i \), therefore the total cost over the spine is \( O(|E_X|) \). Nevertheless, the internal edges \( E'_{X_i} \) of the first bead in common between \( B_{z_i,t} \) and \( B_{z_i-1,t} \) are explored for each back edge \( b_i \).

Although the edges in \( E'_{X_i} \) are in a common bead between \( B_{z_i,t} \) and \( B_{z_i-1,t} \), these edges must be visited. The entry point in the common bead can be different for \( z_i \) and \( z_i-1 \), the DFS tree of that bead can also be different. For an example, consider the case where \( z_i, \ldots, z_j \) are all in the same bead after the removal of \( u \). The bead strings \( B_{z_i,t} \ldots B_{z_j,t} \) are the same, but the roots \( z_i, \ldots, z_j \) of the certificate are different, so we have to compute the corresponding DFS of the first bead \( |j-i| \) times. Note that this is not the case for the other beads in common: the entry point is always the same.

**Lemma 5.22.** The cost \( O(|E_X| + \sum_{i>1} |E'_{X_i}|) \) on a spine of the recursion tree can be amortized to \( O(|E_X|) \).

**Proof.** We can charge the cost \( O(|E'_{X_i}|) \) of exploring the edges in the first bead in common between \( B_{z_i,t} \) and \( B_{z_i-1,t} \) to another node in the recursion tree. Since this common bead is the head of at least one certificate in the recursion subtree of the left child of the \( i \)-th node of the spine. Specifically, we charge the first and only node
in the leftmost path of the $i$th child of the spine that has exactly the edges $E_{X_i}$ as head of its bead string: (i) if $|E_{X_i}| \leq 1$ it corresponds to a unary node or a leaf in the recursion tree and therefore we can charge it with $O(1)$ cost; (ii) otherwise it corresponds to a first node of a spine and therefore we can also charge it with $O(|E_{X_i}|)$. We use this charging scheme when $i \neq 1$ and the cost is always charged in the leftmost recursion path of $i$th node of the spine, consequently we never charge a node in the recursion tree more than once.

Lemmas 5.21 and 5.22 finalize the proof of Lemma 5.13. Fig. 5.8 shows the certificates of bead strings $B_{z_i,t}$ on the left branches of the spine from Figure 5.5.
Chapter 6

Conclusion and future work

This thesis described optimal algorithms for several relevant problems to the topic of listing combinatorial patterns in graphs. These algorithms share an approach that appears to be general and applicable to diverse incarnations of the problem.

We finish this exposition by considering future directions for the work performed in this thesis.

1. The results obtained can have practical implications in several domains. An experimental analysis of the performance of the algorithms would be beneficial continuation of the work performed. We highlight the problem of listing cycles as the first candidate for experimentations.

2. Defining a model of dynamic data structures that support non-arbitrary operations has the potential to become an important tool in this and other topics. One possible starting point could be a stacked fully-dynamic data-structure model. In this model, to apply an operation on an edge $e$ (or vertex $v$) would imply the undoing of every operation done since the last operation on edge $e$ (resp. vertex $v$).

3. A general formulation of the technique, defining the requirements of the pattern and setting of the problem, would allow the application of the technique as a black box. This would likely require some work on the previous point. A possible difficulty in doing so is that the amortized analysis is specific to on the pattern being listed. This difficulty could be circumvented by parameterizing the time complexity of maintaining the dynamic data structure in function of the number of patterns it guarantees to exist in the input.

4. Application to additional problems of listing, such as bubbles, induced paths and holes described in Section 2.2, would likely achieve positive results.

5. Another idea that likely would lead to positive results is an extension of the analysis of the algorithms and their time complexity to take into account
*succinct encodings of the output.* One clear point where this would be useful, would be the on the problem of listing *st*-paths and cycles. In this case, we believe that the algorithm is already optimally sensitive on the size of the front coding of the output.

6. Further investigation of the applications of the technique to *searching and indexing* of patterns in graphs is recommended. Connected to the previous point, succinct representations of the output can be used as an index.
Bibliography

[1] E. Alm and A.P. Arkin. Biological networks. *Current opinion in structural biology*, 13(2):193–202, 2003.

[2] U. Alon. Biological networks: the tinkerer as an engineer. *Science*, 301(5641):1866–1867, 2003.

[3] A. Arenas, A. Fernandez, S. Fortunato, and S. Gomez. Motif-based communities in complex networks. *Journal of Physics A: Mathematical and Theoretical*, 41(22):224001, 2008.

[4] S. Arunkumar and SH Lee. Enumeration of all minimal cut-sets for a node pair in a graph. *Reliability, IEEE Transactions on*, 28(1):51–55, 1979.

[5] D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

[6] F. Barahona, M. Grötschel, M. Jünger, and G. Reinelt. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36(3):493–513, 1988.

[7] J.M. Barnard. Substructure searching methods: Old and new. *Journal of Chemical Information and Computer Sciences*, 33(4):532–538, 1993.

[8] G.J. Bezem and J. van Leeuwen. Enumeration in graphs. Technical Report RUU-CS-87-07, Utrecht University, 1987.

[9] Etienne Birmelé, Pierluigi Crescenzi, Rui A. Ferreira, Roberto Grossi, Vincent Lacroix, Andrea Marino, Nadia Pisanti, Gustavo Akio Tominaga Sacomoto, and Marie-France Sagot. Efficient bubble enumeration in directed graphs. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *SPIRE*, volume 7608 of *Lecture Notes in Computer Science*, pages 118–129. Springer, 2012.

[10] Etienne Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the Twenty-Fourth*
Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, 2013. [12] [22]

[11] J.R.S. Blair and B.W. Peyton. An introduction to chordal graphs and clique trees. Technical report, Oak Ridge National Lab., TN (United States), 1991. [19]

[12] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.U. Hwang. Complex networks: Structure and dynamics. Physics reports, 424(4):175–308, 2006. [18] [45]

[13] B. Bollobás, L. Pebody, and O. Riordan. Contraction–deletion invariants for graphs. Journal of Combinatorial Theory, Series B, 80(2):320–345, 2000. [22]

[14] J.A. Bondy and U.S.R. Murty. Graph theory with applications, volume 290. Macmillan London, 1976. [11] [15]

[15] L.F. Costa, F.A. Rodrigues, G. Travieso, and P.R.V. Boas. Characterization of complex networks: A survey of measurements. Advances in Physics, 56(1):167–242, 2007. [45]

[16] Reinhard Diestel. Graph Theory (Graduate Texts in Mathematics). Springer, 2005. [62] [73]

[17] R.G. Downey and M.R. Fellows. Parameterized complexity, volume 3. springer New York, 1999. [11]

[18] RJ Duffin. An analysis of the wang algebra of networks. Transactions of the American Mathematical Society, pages 114–131, 1959. [20]

[19] L. Euler. The seven bridges of Konigsberg. Wm. Benton, 1956. [11] [15]

[20] Rui A. Ferreira, Roberto Grossi, and Romeo Rizzi. Output-sensitive listing of bounded-size trees in undirected graphs. In ESA, pages 275–286, 2011. [12] [25] [39]

[21] W. Feussner. Uber stromverzweigung in netzformigen leitern. Ann. Physik, 9:1304–1329, 1902. [20]

[22] W. Feussner. Zur berechnung der stromstarke in netzformigen leitern. Ann. Physik, 15:385–394, 1904. [20]

[23] P. Flajolet and R. Sedgewick. Analytic combinatorics. Cambridge University press, 2009. [11]

[24] J. Flum and M. Grohe. Parameterized complexity theory, volume 3. Springer Berlin, 2006. [11]
[25] K. Fukuda, T.M. Liebling, and F. Margot. Analysis of backtrack algorithms for listing all vertices and all faces of a convex polyhedron. *Computational Geometry*, 8(1):1–12, 1997. 21

[26] K. Fukuda and T. Matsui. Finding all the perfect matchings in bipartite graphs. *Applied Mathematics Letters*, 7(1):15–18, 1994. 22

[27] Harold N. Gabow and Eugene W. Myers. Finding all spanning trees of directed and undirected graphs. *SIAM Journal on Computing*, 7(3):280–287, 1978. 25

[28] B. Gavish. Topological design of centralized computer networks—formulations and algorithms. *Networks*, 12(4):355–377, 1982. 19

[29] M. Girvan and M.E.J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002. 18

[30] L.A. Goldberg. *Efficient algorithms for listing combinatorial structures*, volume 5. Cambridge University Press, 2009. 21

[31] M.C. Golumbic. Trivially perfect graphs. *Discrete Mathematics*, 24(1):105–107, 1978. 19

[32] R.L. Graham, P. Hell, and Simon Fraser University. School of Computing Science. *On the history of the minimum spanning tree problem*. Simon Fraser University, School of Computing Science, 1982. 18

[33] I. Gribkovskaia, Ø. Halskau Sr, and G. Laporte. The bridges of Königsberg—a historical perspective. *Networks*, 49(3):199–203, 2007. 11, 15

[34] S. Guillemot and F. Sikora. Finding and counting vertex-colored subtrees. *Mathematical Foundations of Computer Science 2010*, pages 405–416, 2010. 11

[35] SL Hakimi. On trees of a graph and their generation. *Journal of the Franklin Institute*, 272(5):347–359, 1961. 20

[36] T. R. Halford and K. M. Chugg. Enumerating and counting cycles in bipartite graphs. In *IEEE Communication Theory Workshop*, 2004. 63

[37] F. Harary. A characterization of block-graphs. *Canad. Math. Bull*, 6(1):1–6, 1963. 19

[38] F. Harary and E.M. Palmer. Graphical enumeration. Technical report, DTIC Document, 1973. 11 20
[39] M. Hay, G. Miklau, D. Jensen, D. Towsley, and P. Weis. Resisting structural re-identification in anonymized social networks. *Proceedings of the VLDB Endowment*, 1(1):102–114, 2008. 45

[40] T. Hogg and L. Adamic. Enhancing reputation mechanisms via online social networks. In *Proceedings of the 5th ACM conference on Electronic commerce*, pages 236–237. ACM, 2004. 45

[41] Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. Cyclic pattern kernels for predictive graph mining. In *Proc. of 10th ACM SIGKDD*, pages 158–167, 2004. 63

[42] F.K. Hwang, D.S. Richards, and P. Winter. *The Steiner tree problem*, volume 53. North Holland, 1992. 18

[43] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975. 13 22 61 62 63

[44] D.S. Johnson, M. Yannakakis, and C.H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988. 21 22

[45] B.H. Junker and F. Schreiber. *Analysis of biological networks*, volume 1. Wiley Online Library, 2008. 45

[46] Sanjiv Kapoor and Hariharan Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM J. Comput.*, 24:247–265, April 1995. 25

[47] S. Klamt and et al. A methodology for the structural and functional analysis of signaling and regulatory networks. *BMC Bioinformatics*, 7:56, 2006. 63

[48] S. Klamt and A. von Kamp. Computing paths and cycles in biological interaction graphs. *BMC Bioinformatics*, 10:181, 2009. 63

[49] D. Knuth. The art of computer programming, volume 4, 2006. 11 20

[50] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1):1–30, 2001. 22

[51] I. Koutis and R. Williams. Limits and applications of group algebras for parameterized problems. *Automata, languages and programming*, pages 653–664, 2009. 11

[52] D.L. Kreher and D.R. Stinson. *Combinatorial algorithms: generation, enumeration, and search*. CRC, 1998. 21
[53] T.I. Lee, N.J. Rinaldi, F. Robert, D.T. Odom, Z. Bar-Joseph, G.K. Gerber, N.M. Hannett, C.T. Harbison, C.M. Thompson, I. Simon, et al. Transcriptional regulatory networks in saccharomyces cerevisiae. *Science Signalling*, 298(5594):799, 2002. [19]

[54] J. Leskovec. Stanford large network dataset collection, 2012. [15]

[55] Hongbo Liu and Jiaxin Wang. A new way to enumerate cycles in graph. In *AICT and ICIW*, pages 57–59, 2006. [63]

[56] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. *Algorithm Theory-SWAT 2004*, pages 260–272, 2004. [22]

[57] Prabhaker Mateti and Narsingh Deo. On algorithms for enumerating all circuits of a graph. *SIAM J. Comput.*, 5(1):90–99, 1976. [62]

[58] Y. Matsui and T. Matsui. Enumeration algorithm for the edge coloring problem on bipartite graphs. *Combinatorics and Computer Science*, pages 18–26, 1996. [22]

[59] Y. Matsui and T. Uno. On the enumeration of bipartite minimum edge colorings. *Graph Theory in Paris*, pages 271–285, 2007. [22]

[60] J.J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982. [19]

[61] K.L. McMillan. Symbolic model checking: an approach to the state explosion problem. Technical report, DTIC Document, 1992. [19]

[62] E. Meléndez-Hevia, T.G. Waddell, and M. Cascante. The puzzle of the krebs citric acid cycle: assembling the pieces of chemically feasible reactions, and opportunism in the design of metabolic pathways during evolution. *Journal of Molecular Evolution*, 43(3):293–303, 1996. [19]

[63] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002. [19, 45]

[64] G. Minty. A simple algorithm for listing all the trees of a graph. *Circuit Theory, IEEE Transactions on*, 12(1):120–120, 1965. [20, 25]

[65] N. Modani and K. Dey. Large maximal cliques enumeration in sparse graphs. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1377–1378. ACM, 2008. [22]

[66] JW Moon. Counting labelled trees, canadian mathematical monographs, no. 1. In *Canadian Mathematical Congress*, 1970. [20, 25]
[67] C.R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):046116, 2003.

[68] Shin-ichi Nakano and Takeaki Uno. Constant time generation of trees with specified diameter. In Juraj Hromkovic, Manfred Nagl, and Bernhard Westfechtel, editors, *Graph-Theoretic Concepts in Computer Science*, volume 3353 of *Lecture Notes in Computer Science*, pages 33–45. Springer Berlin / Heidelberg, 2005.

[69] P.A. Pevzner, H. Tang, and G. Tesler. De novo repeat classification and fragment assembly. *Genome Research*, 14(9):1786–1796, 2004.

[70] S. Pirzada. Applications of graph theory. *PAMM*, 7(1):2070013–2070013, 2008.

[71] J. Ponstein. Self-avoiding paths and the adjacency matrix of a graph. *SIAM Journal on Applied Mathematics*, 14:600–609, 1966.

[72] J. Propp et al. Enumeration of matchings: problems and progress. *New Perspectives in Geometric Combinatorics* (eds. L. Billera, A. Björner, C. Greene, R. Simeon, and RP Stanley), Cambridge University Press, Cambridge, pages 255–291, 1999.

[73] R C Read and Robert E Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

[74] G. Robertson, J. Schein, R. Chiu, R. Corbett, M. Field, S.D. Jackman, K. Mungall, S. Lee, H.M. Okada, J.Q. Qian, et al. De novo assembly and analysis of rna-seq data. *Nature methods*, 7(11):909–912, 2010.

[75] M. Rospocher. On the computational complexity of enumerating certificates of NP problems. PhD thesis, DIT-University of Trento, 2006.

[76] F. Ruskey. Combinatorial generation. *Preliminary working draft*. University of Victoria, Victoria, BC, Canada, 2003.

[77] K. Sankar and A.V. Sarad. A time and memory efficient way to enumerate cycles in a graph. In *Intelligent and Advanced Systems*, pages 498–500, 2007.

[78] C. Savage. A survey of combinatorial Gray codes. *SIAM review*, 39(4):605–629, 1997.

[79] R. Schott and George Stacey Staples. Complexity of counting cycles using Zeons. *Computers and Mathematics with Applications*, 62:1828–1837, 2011.
[80] M. Schwartz. *Telecommunication networks: protocols, modeling and analysis*, volume 7. Addison-Wesley Reading, Massachusetts, 1987. 19

[81] S.S. Shen-Orr, R. Milo, S. Mangan, and U. Alon. Network motifs in the transcriptional regulation network of escherichia coli. *Nature genetics*, 31(1):64–68, 2002. 19

[82] Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM Journal on Computing*, 26:678–692, 1994. 22 25

[83] J.T. Simpson, K. Wong, S.D. Jackman, J.E. Schein, S.J.M. Jones, and I. Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009. 20

[84] S.H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, 2001. 19

[85] E.H. Sussenguth. A graph-theoretical algorithm for matching chemical structures. *J. Chem. Doc.*, 5:36–43, 1965. 62

[86] Maciej M. Syslo. An efficient cycle vector space algorithm for listing all cycles of a planar graph. *SIAM J. Comput.*, 10(4):797–808, 1981. 62

[87] Jayme L. Szwarcfiter and Peter E. Lauer. A search strategy for the elementary cycles of a directed graph. *BIT Numerical Mathematics*, 16, 1976. 62

[88] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972. 75

[89] Robert Endre Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.*, 2(3):211–216, 1973. 20 62

[90] James C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications ACM*, 13:722–726, 1970. 20 62

[91] S. Tsukiyama, I. Shirakawa, H. Ozaki, and H. Ariyoshi. An algorithm to enumerate all cutsets of a graph in linear time per cutset. *Journal of the ACM (JACM)*, 27(4):619–632, 1980. 22

[92] J.R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976. 19

[93] T. Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. *Algorithms and Computation*, pages 92–101, 1997. 22
[94] T. Uno. A fast algorithm for enumerating bipartite perfect matchings. *Algorithms and Computation*, pages 367–379, 2001. 22

[95] T. UNO. An output linear time algorithm for enumerating chordless cycles. *92nd SIGAL of Information Processing Society Japan*, pages 47–53, 2003. 22

[96] T. Uno. An efficient algorithm for enumerating pseudo cliques. *Algorithms and Computation*, pages 402–414, 2007. 22

[97] Takeaki Uno. New approach for speeding up enumeration algorithms. *Algorithms and Computation*, pages 287–296, 1998. 22

[98] Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms. 2003. 22

[99] L.G. Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979. 21

[100] L.G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. 21

[101] KT Wang. On a new method for the analysis of electrical networks. *Nat. Res. Inst. for Engineering, Academia Sinica Memoir*, (2):19, 1934. 20

[102] Kunihiro Wasa, Takeaki Uno, and Hiroki Arimura. Constant time enumeration of bounded-size subtrees in trees and its applications. *COCOON 2012*. 18 25

[103] X. Wei and Z. Gang. Method for establishing a social network system based on motif, social status and social attitude, November 21 2006. US Patent App. 11/603,284. 45

[104] John T. Welch, Jr. A mechanical analysis of the cyclic structure of undirected linear graphs. *J. ACM*, 13:205–210, 1966. 20 62

[105] S. Wernicke. Efficient detection of network motifs. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 3(4):347–359, 2006. 45

[106] Marcel Wild. Generating all cycles, chordless cycles, and hamiltonian cycles with the principle of exclusion. *J. of Discrete Algorithms*, 6:93–102, 2008. 63

[107] H.S. Wilf and A. Nijenhuis. *Combinatorial algorithms: an update*. SIAM, 1989. 21

[108] S.S. Yau. Generation of all hamiltonian circuits, paths, and centers of a graph, and related problems. *IEEE Transactions on Circuit Theory*, 14:79–81, 1967. 63
[109] D.R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, 2008.