TECHNISCHE HOCHSCHULE MITTELHESSEN

CAMPUS GIESSEN

MNI

Mathematik, Naturwissenschaften und Informatik

Bachelorthesis

Implementation and Evaluation of Space-Efficient Euler Tours and Subgraph Stacks

Submitted in Partial Fulfillment of the Requirements for the Degree
Bachelor of Science (B.Sc.)

to the
Department of MNI
Technische Hochschule Mittelhessen
University of Applied Science
by

Johannes Meintrup

First examiner: Dr. Dr. Frank Kammer
Second examiner: Prof. Dr. Andreas Gogol-Döring

Gießen, November 2018

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, den 14.11.2018

Johannes Meintrup

Abstract

In today's world most algorithms and data-structures focus on runtime efficiency. In fields related to *big data*, space is becoming a crucial issue. In this work we focus on saving space, while maintaining fast runtimes. Many algorithms solve large graph problems recursively, for which we want to provide a framework. We specifically want to focus on creating a foundation to solve bipartite edge-coloring problems in a space-efficient manner. Bipartite edge-coloring is a typical method for solving complex scheduling problems. We implement a space-efficient subgraph stack and a space-efficient Hierholzer algorithm to create Euler partitions (both previously described theoretically in [HKL17]) and various other underlying data-structures needed for such a framework. We conduct runtime and space measurements, comparing our implementations to typical implementations. Our measurements show a large improvement in both runtime and space usage for the subgraph stack implementation and we obtain a superior space-efficiency for the Hierholzer algorithm. Conclusively, our implementation can be used as a foundation of a multitude of recursive graph-algorithms, especially bipartite edge-coloring for which we provide the necessary data-structures. All implementations are included in the library for space-efficient algorithms [Tec18].

German Abstract

In der heutigen Welt sind die meisten Algorithmen und Datenstrukturen auf Laufzeiteffizienz ausgelegt. In Bereichen im Zusammenhang mit *Big Data* wird der steigende Bedarf an Speicherplatz jedoch problematisch. In dieser Arbeit wird der primäre Fokus auf Speicherplatzeinsparungen gelegt, mit dem Ziel, ähnliche Laufzeiteffizienz beizubehalten. Ein platzeffizientes Framework für Algorithmen zum rekursiven Lösen von Graphproblemen wird implementiert. Ein spezieller Fokus liegt auf dem Lösen von Kantenfärbungsproblemen in bipartiten Graphen; eine typische Methode zum Lösen von Scheduling-Problemen. Hierzu wird ein platzeffizienter Subgraph Stack und eine platzeffiziente Version des Hierholzer Algorithmus zum Erstellen von Euler-Partitionen (beide vorherig theoretisch in [HKL17] beschrieben) und weitere grundlegende Datenstrukturen implementiert. Messungen zum Vergleich der Laufzeit und des Platzbedarfs werden durchgeführt. Die Ergebnisse der Messungen zum Subgraph Stack zeigen eine verbesserte Laufzeit und Platzeffizienz. Die Ergebnisse der Messungen zum Hierholzer Algorithmus zeigen eine verbesserte Platzeffizienz. Zusammenfassend kann gesagt werden, dass grundlegende Strukturen für platzeffiziente, rekursive Graphalgorithmen implementiert wurden, die nun als Framework zum Lösen vieler Probleme verwendet werden können. Insbesondere zum platzeffizienten Lösen von Kantenfärbungsproblemen in bipartiten Graphen, für welches nun alle nötigen Datenstrukturen zur Verfügung stehen. Alle implementierten Algorithmen und Datenstrukturen wurden in die Bibliothek für platzeffiziente Algorithmen [Tec18] integriert.

Contents

Eidesstattliche Erklärung i

Abstract iii

German Abstract v

Contents vii

List of Figures ix

1 Introduction 1

1.1 Context . . . 1
1.2 Research Questions and Goals . . . 2
1.3 Strategy . . . 3
1.4 Structure . . . 3

2 Fundamentals and Motivation 5

2.1 Glossary . . . 5
2.1.1 Space Efficiency . . . 5
2.1.2 Graph . . . 6
2.1.3 Subgraph . . . 6
2.1.4 Bipartite graph . . . 7
2.1.5 Euler trail, Euler partition and Euler tour . . . 7
2.1.6 Edge coloring . . . 8
2.1.7 Dyck word . . . 9
2.2 Motivation . . . 9

3 Implementation 13

3.1 Table Lookup . . . 13
3.2 Bitset . . . 13
3.3 Graph . . . 14
3.4 Rank-select . . . 15
3.5 Choice Dictionary . . . 18
3.6 Dyck-matching Structure . . . 19

3.7 Space-efficient Hierholzer Algorithm . . . 22
3.8 Subgraph Stack . . . 28

4 Analysis **37**
4.1 Methodology . . . 37
4.1.1 Setting . . . 37
4.1.2 Measuring runtime and space usage . . . 37
4.2 Measurements . . . 39
4.2.1 Bitset . . . 39
4.2.2 Rank-select . . . 40
4.2.3 Hierholzer algorithm . . . 42
4.2.4 Subgraph stack . . . 45
4.2.5 Combining the Hierholzer algorithm and subgraph stack structure . 48
4.3 Conclusion . . . 51

5 Summary **55**

Bibliography **57**

List of Figures

2.1 Undirected graph example . . . 6
2.2 Subgraph example . . . 6
2.3 Connected component example . . . 7
2.4 Bipartite graph example . . . 7
2.5 Euler trail in a graph . . . 8
2.6 Euler partition of a graph . . . 8
2.7 Euler tour of a graph . . . 8
2.8 Edge-colored bipartite graph . . . 9
2.9 Dyck word and matching parentheses . . . 9
2.10 Modeling a schedule using bipartite edge-coloring . . . 10
2.11 Recursive strategy used for bipartite edge-coloring . . . 11
2.12 Example of modeling a scheduling problem . . . 12

3.1 A graph and the corresponding adjacency arrays . . . 15
3.2 Table based rank-select structure . . . 16
3.3 Space-efficient rank-select structure with an example of $rank$ . . . 17
3.4 Space-efficient rank-select structure with an example of $select$ . . . 18
3.5 Segmentizing a dyck word into blocks . . . 19
3.6 Finding local matches in a segmentized dyck-word . . . 21
3.7 Dyck word showing the pioneers for each block . . . 21
3.8 Depiction of the steps of the Hierholzer algorithm . . . 23
3.9 Dyck-matching structure created for a trail-structure in phase two . . . 25
3.10 Diagram showing four states of a trail structure D_v . . . 27
3.11 Example for isomorphism from G to G' . . . 28
3.12 The arc-numbering function g for a graph G . . . 30
3.13 The bitsets created for the functions ϕ and ψ when pushing a graph $G_{\ell+1}$ . . . 34
3.14 Three different schemes for choosing a reference graph for a subgraph and their resulting structure . . . 35

4.1 Plot showing the runtime of initializing a bitset . . . 39
4.2 Plots showing the runtime and space requirements of initializing rank-select structures . . . 41
4.3 Plots showing the runtime of the $rank$ and $select$ functions . . . 41

4.4 Plot showing the runtime of the naive and space-efficient Hierholzer algorithm 43
4.5 Plot showing the space requirements of the naive and space-efficient Hierholzer algorithm 44
4.6 Plot showing the runtime of the naive and space-efficient subgraph stacks . . . 46
4.7 Plot showing the space requirements of the naive and space-efficient subgraph stacks 47
4.8 Plot showing the runtime of the naive and space-efficient implementations of the algorithm described in Section 4.2.5 49
4.9 Plot showing the space requirements of the naive and space-efficient implementations of the algorithm described in Section 4.2.5 50

Chapter 1

Introduction

In this chapter we will describe what we want to achieve with this thesis. We will define our goals and the strategy we follow to achieve them.

1.1 Context

Most people have heard the word *algorithm* at least a few times during their life. Through the media the term 'algorithm' has been made known by the public. It has been used to describe everything computer science related from AI to cloud computing in the news. But what is an algorithm? It is a set of instructions to solve a problem. If one follows those instructions exactly, the problem the algorithm is supposed to solve, will be solved. At its core many things we encounter daily are algorithms. A cookbook for example is just a collection of algorithms, and so is the manual to build an *IKEA* cupboard. What we are interested in this work are algorithms in the context of computer science. Algorithms are one of the core concepts in computer science, both in academia and in industry applications. Every program uses algorithms for its internal logic to solve specific problems. The quality of an algorithm can be evaluated from multiple angles. The most common being the *runtime* of an algorithm: How fast does the algorithm solve the problem? This question has been asked the most by researchers and software developers across the globe. The second most common question is: How much *space* does the algorithm need? It is a question that often gets neglected, because speed is the most important factor in most applications. Many students of computer science get told in their first year that space is cheap and plays a secondary role to runtime or cost of development. For many applications this is true. However, in fields related to big data space can become an issue. For instance, when an algorithm uses a lot of space, there is often a slowdown in runtime. This is mainly due to the fact that if the amount of memory which a program uses becomes so monumental, there is a heavy slowdown in runtime associated to memory managing processes by the computer system. Additionally, such algorithms designed to work on massive datasets require large servers to run. Using a solution that focuses on a fast runtime can result in the need for upgrading the hardware of the system. Yet, increasing the amount of available memory on such a server

is not as cheap as increasing the amount of space available on a desktop PC or laptop and often not feasible. It therefore stands to the reason to solve such problems focusing on space saving approaches.
Other situations in which space is a scarce resource is the field of embedded devices. Often these devices are tiny and have a very limited amount of space available. Ordinarily it is too expensive or physically impossible to add more space to these devices. An example for such a device would be a tiny micro-controller using 16 kByte of working memory, an amount magnitudes smaller than the size of a typical photograph. Doubling the space on such a small device can result in tripling, or even quadrupling the price [rei18a, rei18b]. Many typical algorithms that are widely used were not developed with a focus of saving space in mind. A software developer who wants to implement such solutions will run into a problem. There are no established programming libraries for algorithms focusing on minimizing space-usage in any programming language available. This implies the need to create a framework for various algorithms optimized for space usage.

1.2 Research Questions and Goals

The goal of this work is the implementation and analysis of a set of data-structures and algorithms optimized for space-usage, more precisely that satisfy the definition of being space-efficient and can be used as a framework for recursive algorithms working on graphs. For more details on the framework see Section 2.2 and Chapter 3. The runtime and space requirements of these structures and algorithms should be analyzed via runtime tests and profiling.
The implementation should be integrated and documented in the library for space-efficient algorithms [Tec18], which is being developed in parallel to this work.

The following questions should be answered in detail:

1. What does the implementation look like? What do we need to implement the complete framework for recursion and how do we implement the 'trivial' details? (Chapter 3)

2. What changes have to be made while making the transition from the theoretical structures to the implementation? Are there any differences between the theoretical and the practical? (Chapter 3)

3. How efficient is our implementation regarding runtime and space compared to standard structures and algorithms? (Chapter 4)

4. How can a real-world problem be solved with the use of our implementation? (Section 2.2 and Section 4.3)

1.3 Strategy

Our goal is to implement the framework proposed in [HKL17]. To do this, we need to analyze exactly what is needed for the algorithm. First we take apart the framework to dissect the underlying structures that are needed. An algorithm that needs to be space-efficient has to be space-efficient at every part, meaning each underlying structure needs to be space-efficient. The first step is implementing these structures.
The framework needs to be understood from a top-down view before any implementation can occur. Once that is done it is possible to implement the structures from bottom to top, i.e., starting at the most basic structures. Usually, the implementation details for the structures that are being described in theoretical works are only sketched roughly. These 'trivial' details have to be solved for a practical implementation. Once we have implemented everything that is needed we can create the framework. Following that we have to analyze it and profile the space and time requirements. At the end we answer the question: does the framework work as efficiently as intended?

1.4 Structure

The subsequent chapters are structured in the following way: we start by introducing some basic concepts and keywords. These are needed for the further understanding of the work and in later chapters it is assumed that the reader is familiar with them. We follow by describing our motivation for the work and describing some use cases and real life examples of what we want to achieve with the implementation. The next chapter will describe the implemented structures and outlines the differences between the theoretical and the practical. After that follows the analysis of the implementation, which starts by describing our methods for profiling and the results. Then we examine and evaluate the measurements. At the end a brief summary will conclude the work.

Chapter 2

Fundamentals and Motivation

In this chapter we focus on the fundamentals our work is based on. In the glossary we describe some keywords and structures that need to be known to understand this work. Following that is the motivation chapter where we describe some practical applications of our work.

2.1 Glossary

2.1.1 Space Efficiency

To understand *space efficiency* we first have to define what space with the relation to algorithms and data structures means. When we talk about space, we are referring to the size of the data structures, usually in bits, that is needed to store them in the random access memory. We want to be efficient in regards to working memory. For many of the structures and algorithms the size of the input is very large and does take up a lot of space. We do not want to minimize the size of the input, but the size of the structures and algorithms that work on such large inputs. We denote space needed with the *Big-O Notation*. When considering to minimize the size requirements of an algorithm it is mandatory to minimize the size requirements of the underlying data-structures used by the algorithm. With an understanding of what space means in regards to data-structures and algorithms, we can now define the term *space efficiency*. If an algorithm or data structure needs s bits of working memory and s is small compared to the requirements of comparable structures or algorithms—usually the industry standard—we call them *space-efficient*. It is usually also needed that the runtime of space-efficient algorithms and data structures is similar to that of the standard. The most naive way to implement a space-efficient algorithm would be to never save any data unless it is immediately needed and constantly re-calculate everything. This, of course, is not what the aim of space efficiency is, since it would add a significant overhead in runtime. Instead we usually talk about having a smaller overhead of space used than the standard structures while maintaining similar runtimes and especially similar runtime growths. I.e., when the industry standard for some algorithm runs in $O(n)$ time and uses $O(n \log n)$ bits space, a space-efficient version of that algorithm might run in $O(n)$ time and use $O(n)$ or $O(n \log \log n)$ bits of space.

2.1.2 Graph

A graph $G = (V, E)$ is composed of a set of vertices $V = \{1, \dots, n\}$ and a set of edges $E = \{1, \dots, m\}$. An edge $e \in E$ connects two vertices $v, u \in V$, called the *endpoints* of e. An edge can be *directed* or *undirected*. Informally, a directed edge can only be traversed in one direction, while an undirected edge can be traversed in both directions. We call a vertex v and an edge e *incident* with each other if e has v as one of its endpoints. Two edges e and e' with $e \neq e'$ are *adjacent* if they share a vertex v as an endpoint. We call $|V| = n$ the *order* of a graph. The number of adjacent edges a vertex v has is called the *degree* of v. The degree of the vertex with the largest number of adjacent edges is denoted by Δ (largest degree). See Figure 2.1 for a visual example.

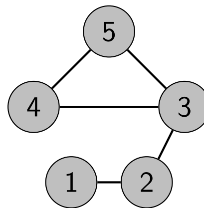

Figure 2.1: An undirected graph $G = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (3, 5), (3, 4), (3, 2), (4, 5)\}$.

2.1.3 Subgraph

Let $G = (V, E)$ be a graph. We call $G_{sub}(V_{sub}, E_{sub})$ a *subgraph* of G if $E_{sub} \leq E$, and G the *supergraph* of G_{sub}. The set V_{sub} contains at least all endpoints of the edges contained in E_{sub} (see Figure 2.2). It is also possible that vertices with a degree of zero are contained in V_{sub}. A *connected component* of an undirected graph G is a subgraph G_{sub} in which any two vertices $v,\ u \in V_{sub}$ are connected to each other with paths, which are not connected to any additional vertices in the supergraph G (see Figure 2.3).

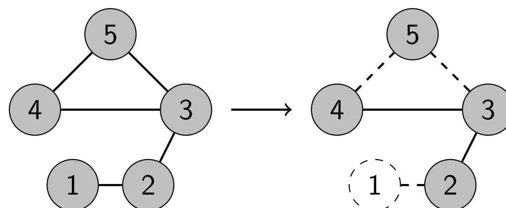

Figure 2.2: On the left is the undirected graph $G = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (3, 5), (3, 4), (3, 2), (4, 5)\}$. On the right is the subgraph G_{sub} of G with $V_{sub} = \{2, 3, 4, 5\}$ and $E_{sub} = \{(3, 4), (3, 2)\}$. The dashed edges and vertices in the second graph represent the edges and vertices that are not in V_{sub} or E_{sub}.

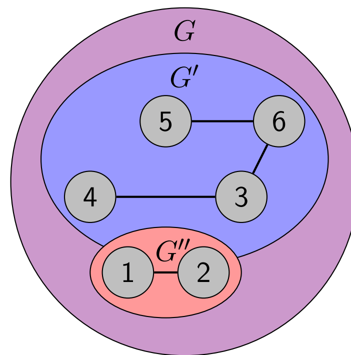

Figure 2.3: A graph G consisting of two connected components with $G' = (V', E')$ with $V' = \{3, 4, 5, 6\}$ and $E' = \{(3, 4), (3, 6), (5, 6)\}$ and $G'' = (V'', E'')$ with $V'' = \{1, 2\}$ and $E'' = \{(1, 2)\}$.

2.1.4 Bipartite graph

We call a graph *bipartite* if its set of vertices V can be divided in two independent sets U and W in such a way that for every edge $e \in E$, e can only have exactly one endpoint in U and one endpoint in W. Informally, every edge connects a vertex from U to a vertex in W, but never from U to U or from W to W. See Figure 2.4 for an example.

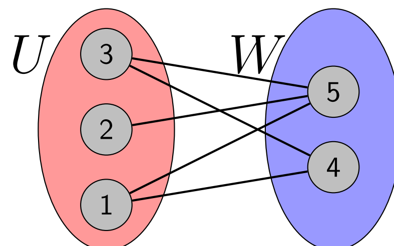

Figure 2.4: Bipartite graph with two independent sets of vertices $U = \{1, 2, 3\}$ in red and $W = \{4, 5\}$ in blue.

2.1.5 Euler trail, Euler partition and Euler tour

Take an undirected graph $G = (V, E)$ with $E \neq \emptyset$. We call an *Euler trail* a walk through G traversing the edges without using an edge more than once. To start a trail we choose an arbitrary vertex v. Then we choose an arbitrary unmarked incidental edge e_1 of v and mark it. Traversing e_1 to its endpoint we arrive at a vertex u. Again, we choose an arbitrary unvisited incidental edge e_2, mark it, and follow it to its endpoint. We repeat this process until we arrive at a vertex with no unmarked incident edges. The sequence of these traversed edges define an Euler trail (Figure 2.5). If the first vertex in the trail is also the last vertex in that trail, we call the trail *closed*. Otherwise it is called *open*. We can create such Euler trails for directed or undirected graphs, but Euler trails itself are always directed. The endpoint

of an edge in the trail is always the starting vertex of the next edge in the sequence. We call a set of *Euler trails* of minimal cardinality in G, such that all edges of G are visited, an *Euler partition* of G (see Figure 2.6). A graph with a partition consisting of exactly one closed trail is called *Eulerian*, and if the trail is not closed, *semi-Eulerian*. When an Euler partition consists of exactly one closed Euler trail, we call that trail an *Euler tour* (Figure 2.7). Note that the Euler trails of this collection form a partition of E. The number of Euler trails contained in a partition for each connected component G' of a graph G with k vertices of uneven degree in that component is $\lceil k/2 \rceil$ [HKL17].

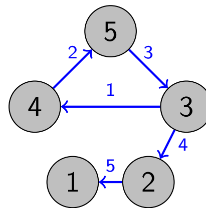

Figure 2.5: An Euler trail in a graph. The trail is shown in blue. There are many possible trails that can be created for this graph, but the trails always visit every edge.

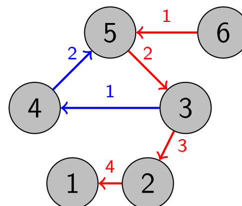

Figure 2.6: An Euler partition for the given graph. This graph can be partitioned into exactly two Euler trails, with one trail shown in red, and the other trail shown in blue. The partition is not distinct, but there are always exactly two trails in a partition for this graph, since there are $k = 4$ vertices of uneven degree.

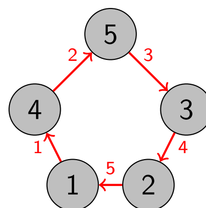

Figure 2.7: An Euler partition for the given Eulerian graph. Since the graph is Eulerian, the Euler partition only contains one closed Euler trail, which we call an Euler tour.

2.1.6 Edge coloring

Edge coloring of a graph is an assignment of *colors* to the edges of the graph in a way that two adjacent edges do not share the same assigned color. Finding a solution to the problem of coloring a graph with at most k different colors is called an *edge-coloring problem*. A

specific version of the edge-coloring problem is edge coloring for a bipartite graph. In a bipartite graph G the maximum number k of colors needed is always Δ [Kön16]. See Figure 2.8 for an example.

2.1.7 Dyck word

A *dyck word* is a word over the alphabet $\Sigma_D = \{(,)\}$ that can be generated by the context-free grammar $S \rightarrow \epsilon \mid (S) \mid SS$. Informally, a dyck word is a properly nested sequence of parentheses. The number of closing parentheses is always the same as the number of opening parentheses. Let d be a dyck word of length $n \geq 2$. Each opening parenthesis o of d has exactly one closing parenthesis c in d as its *match* (Figure 2.9). Two matching parentheses are always created with the grammatical rule $S \rightarrow (S)$.

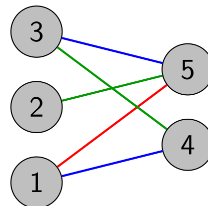

Figure 2.8: Edge coloring in a bipartite graph using $\Delta = 3$ colors.

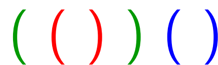

Figure 2.9: Dyck word with matching parentheses sharing the same color.

2.2 Motivation

The goal of this work is the implementation and analysis of a set of data structures that satisfy the definition of being space-efficient and can be used as a framework for recursive algorithms working on graphs. What does it mean to solve a graph problem recursively? Usually, an algorithm that employs a recursive strategy to solve a problem, splits the problem into smaller subproblems, which are again split into even smaller subproblems, until they can be solved easily. These solutions for the subproblems are then combined to a single solution for the larger main problem. For algorithms working on graph structures this often means that each subproblem works on a subgraph of the original graph, which are subsequently combined to a solution for the original graph. When solving graph problems recursively, space can become a major issue. Creating new graph objects at every step of the recursion can quickly result in sizes that are not manageable. This is why we chose to set our focus on the space-efficient representation and management of subgraphs. This should create the foundation for many algorithms solving graph problems recursively. Also, we want to

create the means to solve a concrete problem: Scheduling. A scheduling problem can be modeled as an edge-coloring problem in a bipartite graph. A simple example can be seen in Figure 2.10. Typical problems like creating time tables for university classes, shifts at a

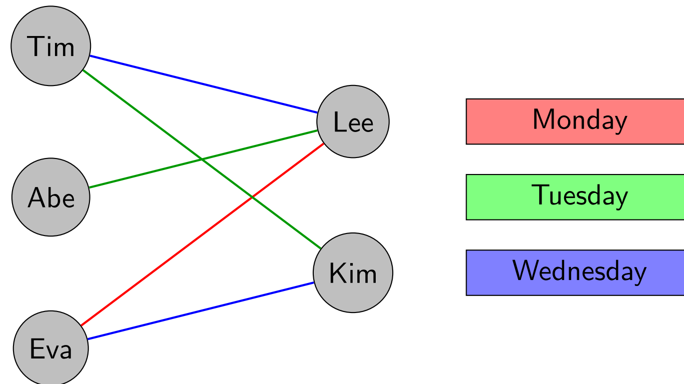

Figure 2.10: Modeling a schedule as a bipartite graph with Tim, Abe and Eva (left) have meetings with Lee and/or Kim (right). Since nobody can be at two places at once, each person can only conduct one meeting each day. The edge-colored graph represents a solution to this scheduling problem. Each edge color states the day where the meeting takes place.

workplace or scheduling meetings are not complex enough to need space-efficient solutions. Nonetheless, there are a great many use cases where the graph structures become too large for common approaches. Taking the *Port of Hamburg* as an example, which is the third largest port in Europe. One can imagine numerous logistic problems arise at such a large facility. At the port, there are over 90 million tons of containers that are being transferred a year [**?**]. Solving a variety of scheduling problems is a daily occurrence. No ship can be at two places at once, and neither can a container. Goods have to be scheduled to containers, containers have to be scheduled to ships, and ships have to be scheduled to routes and docks. Cranes and other machinery that transport the containers have to be scheduled. The weight of the containers has to be balanced on the ships, so there is no lopsided weight distribution, and many more such things have to be considered. And these are just the problems at a single port. Ships and goods have to be scheduled for the entire network of ports world wide. An example of a concrete (simplified) scheduling problem related to the world-wide network of ports would be as follows: we have goods that regularly have to be transported from one port in the world to another. Every port in the world can deload a ship in a single hour, but only one ship at a time. For simplicities sake, ships have no traveling time. To find the minimum number of hours needed for all ships to deliver and deload their goods we can model the problem as an edge-coloring problem for a bipartite graph, which can then be solved algorithmically [Gab76] as follows: after creating an Euler partition P for a graph G, we traverse the partition, and add the traversed edges alternately to two subsets A and B. This splits the partition into two roughly equally sized sets. We create a subgraph for each of those sets, G_A and G_B, and recursively create Euler partitions for those graphs, splitting the problem into two smaller problems each time. The recursion

stops when Δ becomes 1 for G_A and G_B respectively, at which point all edges in each subgraph can be colored with a single color and we only need to merge the solutions for the subgraphs back to the original graph G. The algorithm has a runtime of $O(n \log m)$ and the space requirements are dependent on the (sub) graph representation, the representation of the Euler partitions and Δ. A sketch of the algorithm can be seen in Figure 2.11.

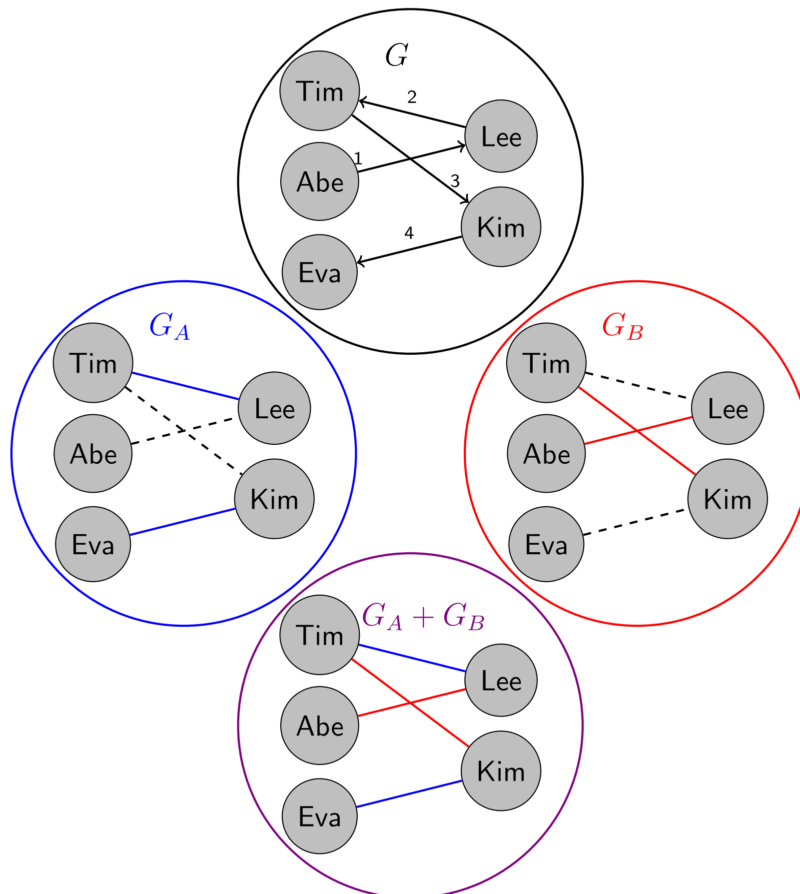

Figure 2.11: After creating an Euler partition for the graph G the problem can be split into two smaller subproblems by creating subgraphs. The subgraphs are created by traversing the trail, and alternately putting the edges into one or the other subgraph (G_A and G_B). Since Δ_A and Δ_B both equal 1, the coloring can be solved directly for G_A and G_B. The next and final step is to combine the solutions into a solution for the supergraph G.

Now, how can we use this algorithm to solve the scheduling problem described previously? For every fully loaded ship that has to transport its goods from one port A to another port B we have an edge $e = (A, B)$. The color of the edge e represents the time of day the ship can be deloaded at the port. An example of a model of this scheduling problem can be seen in Figure 2.12. In reality, such a problem would of course be much more complex. Not

every ship has the same capacity or speed. Maybe ships need to have regular maintenance taken every few tours. Not all goods have the same weight or even shelf life; maybe some perishable goods have a higher priority to be delivered and need to be delivered before a certain date. Also, a port has multiple docks for loading and deloading, all of which have different specifications. A container can also be transported via train or truck, which have to be loaded and scheduled as well. With such large graph problems in mind, we can easily see the need for space-efficient solutions. This is why we want to provide the basic structures and algorithms to solve such graph problems, with a focus on bipartite edge-coloring.

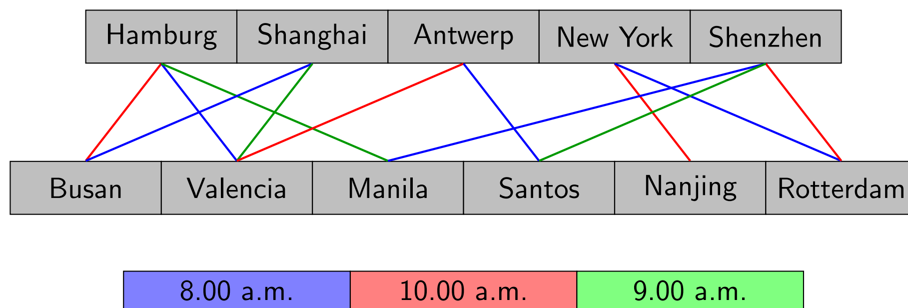

Figure 2.12: An example of a scheduling problem related to the world-wide network of ports. We have a set of goods that regularly have to be transported from one port in the world to another. Every port can load or deload a ship in a single hour, but only one ship at a time. For simplicities sake, ships have no traveling time. Goods have to be transported from the ports at the top row to the ports at the bottom row. The edges represent a set of goods being transported by a ship. Since $\Delta = 3$, it is possible to color the graph using a maximum of three colors. This results in a minimum number of three time-slots to load, transport and deload all goods without waiting times.

Chapter 3

Implementation

In this chapter we will discuss the implementation of the algorithms and structures. We describe all the structures implemented and employed. Unless specifically mentioned, everything in this chapter has been implemented by the author. Most of the implementation is based on the theoretical work layed out in [HKL17], if this is not the case, it is specifically mentioned. We first start by describing the most basic structures used, which are needed for the later understanding of other structures. At the core of the implementation is the *Space-efficient Hierholzer algorithm* (Section 3.7) for creating space-efficient Euler partitions and the *subgraph stack* (Section 3.8) to manage and store subgraphs. The reason for focusing on implementing these two structures can be found in Section 2.2. All structures described in this chapter are contained in the library for space-efficient algorithms [Tec18] and released under the GNU General Public License. The library is written in the programming language C++, with the version C++11 in mind.

3.1 Table Lookup

Table lookup is a simple strategy that works as follows: for some function f compute all values of an input and store them in a table. This way evaluating f for all precomputed values can be done in constant time by looking up the appropriate value in the table. We employ the strategy of table lookup at various points in the implementation. We evaluate some function f for very small domains, usually with 2^8 or 2^{16} elements and store them in an array taking $O(2^8 w)$ and $O(2^{16} w)$ space respectively, with w being the word length. This is done statically so only one table for a given function exists globally during the run of a program. For this reason we usually view the size of a table lookup structure as constant.

3.2 Bitset

The core of most of the implemented structures are *bitset* structures. A bitset is an array of bits of fixed length. Each bit can be addressed individually and manipulated. The structure uses $O(n)$ bits space. All bit operations take constant time. Unfortunately, on a typical machine it is not possible to allocate a single bit of space. The minimum amount of bits

that is able to be addressed is a *byte*. The user will be able to address a single bit, but internally the bitset will use an array of numeric types. The user can choose the numeric type for the internal storage. The choices are words of 8, 16, 32 and 64 bits respectively. We will refer to the chosen word as the *block type* of the bitset. The user can set and read the bits via the *subscript operator '[ ]'*, just as one would with a regular array. For table lookups we often want to read entire blocks of bits as our lookup parameter. These will typically be of the length of the block type. For this reason we also implement getters and setters for the internal blocks.

3.3 Graph

All our implementations that operate on graphs are implemented to operate on undirected graphs. Therefore, our graph implementation is only for undirected graphs. Let $G = (V, E)$ with $V = (1, \ldots, n)$ and $E = (1, \ldots, m)$ with $u, v \in V$ and $e \in E$. We represent each undirected edge $e = \{u, v\}$ of the graph by two directed edges $a = (u, v)$ and $b = (v, u)$ that we call $arcs$, with a being the *mate* of b and vice versa. Our graph consists of an array of vertices with each vertex being represented by an array of its arcs represented by *adjacency* objects. An adjacency object consists of a tuple (u, i) with u being the vertex the arc points to and i being the index the $mate$ has in the adjacency array of the uth vertex called the *cross-index*. An example can be seen in Figure 3.1. The implementation provides the following functions, for which we use the umbrella term *simple graph-queries*:

- $order$ - returning the order of the graph.
- $degree(u)$ - returning the degree of the uth vertex in the node array.
- $head(u, k)$ - returning the head of the kth arc in u's adjacency array.
- $mate(u, k)$ - returning the mate of the kth arc in u's adjacency array.

This graph representation uses $O((|V| + |E|)w)$ bits space, with w being the word length, and all functions can be evaluated in constant time.

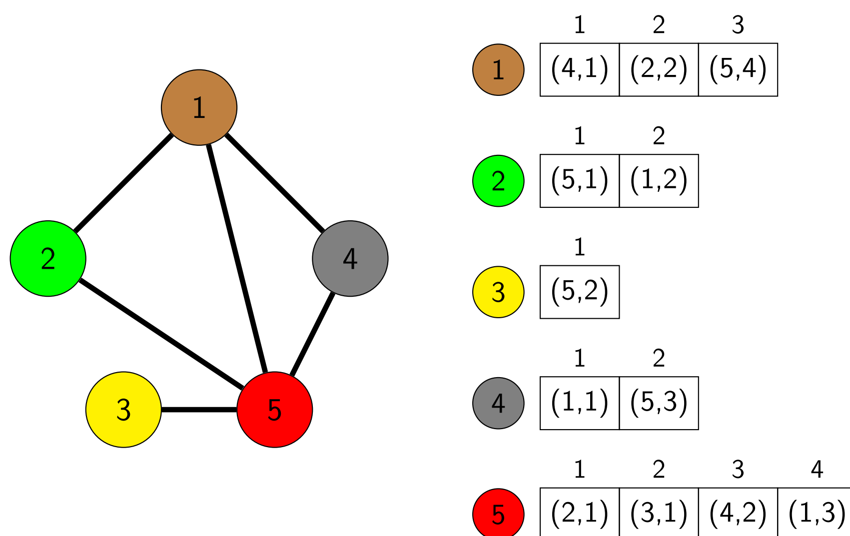

Figure 3.1: A graph with the corresponding adjacency arrays. The entries in the arrays on the right consist of tuples (u, i) with u being the vertex the arc points to, and i being the index of the $mate$ of (u, i) in the adjacency array of the vertex u. The numbers above the arrays represent the indices of the entries. E.g., the third adjacency object of vertex 1 is $(5, 4)$; thus the mate of $(5, 4)$ can be found at index 4 in the adjacency array of vertex 5.

3.4 Rank-select

We now take a look at the *rank-select structure* employed by multiple data-structures in this work. We start by defining the operations of rank-select. Let $B = (b_1, \ldots, b_n)$ be a bitset we call the *input sequence*. For $0 \leq j \leq n$ and $1 \leq k \leq \sum_{i=1}^{n} b_n$ we define the functions $rank_B(j) = \sum_{i=1}^{j} b_i$ and $select_B(k) = min\{j \mid 1 \leq j \leq n \wedge rank_B(j) = k\}$. A rank-select structure is a structure that enables us to evaluate $rank_B$ and $select_B$ in constant time.
The naive approach is to simply iterate over the input sequence and store the values for $rank_B$ and $select_B$ in a table (Figure 3.2). When storing the values as words of length $\log n$ bits this would occupy $O(n \log n)$ bits space. What is needed is a structure that enables us to lookup the values in constant time, but occupies less space than a simple table structure.
Our approach starts by segmentizing the input sequence $B = (b_1, \ldots, b_n)$ into segments of length $r = \lfloor (\log n)/2 \rfloor$, except for the last segment which can be smaller. This segmentizes B into $q = \lceil n/r \rceil$ segments, which we call $blocks$. For $k \in 1, \ldots, q$ we create a table A with $A[k] = \sum_{i=1}^{k-1} \sum_{j=1}^{r} b_{((i-1)*r+j)}$. Informally, $A[k]$ stores the number of set bits before the kth block. If the structure should support $select$ as well we create two additional structures. The first is a table M, which holds the information of $M[k] =$ index of the kth non-empty block. The second is a bitset F setup as follows: a bit at index k in F is set to 1 precisely if the kth bit in B is the first bit set in its block. For F we also create a rank-structure which consists of the table A_F. There is no F_F or M_F needed since we only need to evaluate $rank_F$.

	1	2	3	4	5	6	7	8	9	10	11	12
B:	0	0	1	0	1	1	0	1	0	0	0	1

	1	2	3	4	5	6	7	8	9	10	11	12
Rank$_B$:	0	0	1	1	2	3	3	4	4	4	4	5

	1	2	3	4	5
Select$_B$:	3	5	6	8	12

Figure 3.2: Rank-select structure represented by a simple table lookup for $select$ and $rank$.

The last structures we need to create are the lookup tables T_R to evaluate $rank$ and T_S to evaluate $select$ for our small blocks. Let $B' \in \{0,1\}^r$ and $0 \leq j \leq r$, then $T_R[B', j] = rank_{B'}(j)$ and $T_S[B', j] = select_{B'}(j)$. In our implementation T_R and T_S is implemented as a statically allocated array with the values calculated for all possible B's of a certain length: a strategy described in Section 3.1. When talking about evaluating T_R and T_S we say we evaluate the *local* $rank$ or $select$ values, otherwise we refer to them as *global* values. In our implementation our blocks always have a length of 8 bits. This way only one table has to be created for any amount of rank-select objects that are created during runtime, and the lookup is faster, because this way the blocks created by segmentizing the input sequence B are the exact same blocks used for the internal storage as described in Section 3.2. Since our lookup tables exist only once, no matter how many rank-select objects we create, the space overhead can be viewed as negligible. T_R and T_S can be evaluated in constant time after initialization.
The only thing that is missing now, is how do we evaluate $rank_B$ and $select_B$ in constant time? Once initialized, $rank_B$ works as follows:

$$rank_B(j) = A[\frac{j-1}{r} + 1] + T_R[B', ((j-1) \bmod r) + 1]$$

with B' being the $[\frac{j-1}{r} + 1]$th block. Informally, $rank_B(j)$ is reduced to two function calls. First, we evaluate how many bits are set before the block that contains j. Since each block is of length r, this can be obtained via $A[\frac{j-1}{r}]$. Now we only have to find out how many bits are set before j in its own block, which we can evaluate via table lookup. Both parts of the function are evaluated in constant time and just have to be added. An example can be seen in Figure 3.3. The function $select$ can be evaluated in a similar way:

$$\mathbf{h} = M[rank_F(k) + 1]$$
$$select_B(k) = T_S[B'_h, k - A[\mathbf{h}]] + r(\mathbf{h} - 1)$$

The variable h equals the number of the block with the kth set bit in B. Therefore $A[h]$ equals the number of set bits before the hth block. $k - A[h]$ delivers us k_ℓ which is our

	1	2	3	4	5	6	7	8	9	10	11	12
B:	0	0	1	0	1	1	0	1	0	0	0	1
	1	2	3	4	1	2	3	4	1	2	3	4
	B'_1				B'_2				B'_3			

	1	2	3
A:	0	1	4

$$rank_B(6) = A[\tfrac{6-1}{4} + 1] + T_R[\mathtt{1101}_2, ((6-1) \bmod 4) + 1]:$$
$$rank_B(6) = A[2] + T_R[\mathtt{1101}_2, ((6-1) \bmod 4) + 1]$$
$$rank_B(6) = 1 + T_R[\mathtt{1101}_2, ((6-1) \bmod 4) + 1]$$
$$rank_B(6) = 1 + T_R[\mathtt{1101}_2, 2]$$
$$rank_B(6) = 1 + 2$$
$$rank_B(6) = 3$$

Figure 3.3: Space-efficient rank-select structure with an example how $rank$ works. In this example the size of the blocks is $r = 4$ for a bitset $B = (b1, \ldots, b_n)$ with $n = 12$. The indices the bits have in their local blocks can be seen below the bitset B. The table A is created for each rank-select structure, while the table T_R exists globally and the lookup works as in our previous example.

parameter for our local select table lookup for the block B'_h. Now we have selected the k_ℓth bit in B'. We only have to add the second part of the equation, $r(h-1)$, which equals the number of bits in B before the hth block. An example can be seen in Figure 3.4. Informally, for both $rank$ and $select$, there is always a local evaluation for the block B' and then we have to add a value for all the blocks that came before B', which is evaluated globally.

With the structures initialized we can now evaluate $rank$ and $select$ in constant time. Initialization takes $O(n)$ time. If we discount the space used by the bitset $B = (b_1, \ldots, b_n)$ the rank-structure for B uses $O((n/r)w)$ bits for A with w being the word-length used for storage. For supporting $select$ we need $O((n/r)w)$ bits for M and $O(n + (n/r)w)$ bits for F. With the optimal case being $r = \Theta(\log n) = \Theta(w)$ resulting in $O(n)$ bits of used space.

	1	2	3	4	5	6	7	8	9	10	11	12
B:	0	0	1	0	1	1	0	1	0	0	0	1
	1	2	3	4	1	2	3	4	1	2	3	4
		B'_1				B'_2				B'_3		

	1	2	3	4	5	6	7	8	9	10	11	12
F:	0	0	1	0	1	0	0	0	0	0	0	1
$rank_F$:	0	0	1	1	2	2	2	2	2	2	2	3

	1	2	3
A:	0	1	4

	1	2	3
M:	1	2	3

$$select_B(4) = T_s[B'_h, 4 - A[\mathbf{h}]] + 4(\mathbf{h} - 1)$$

$$\mathbf{h} = M[rank_F(4) + 1] = M[1 + 1] = M[2] = 2$$

$$select_B(4) = T_s[B'_2, 4 - A[2]] + 4 \cdot 1$$

$$select_B(4) = T_s[1101_2, 3] + 4 \cdot 1$$

$$select_B(4) = 4 + 4 \cdot 1$$

$$select_B(4) = 8$$

Figure 3.4: Space-efficient rank-select structure with an example how $select$ works. In this example the size of the blocks is $r = 4$ for a bitset $B = (b1, \ldots, b_n)$ with $n = 12$. The indices the bits have in their local blocks can be seen below the bitset B. The table A, M and the rank structure F is created for each rank-select structure. The table T_S exists as a singleton structure.

3.5 Choice Dictionary

A *choice dictionary* is a data type that, for arbitrary $n \in \mathbb{N}$ can be initialized with a call $init(n)$ and subsequently maintains an initially empty subset $I = 1, \ldots, n$, which we call its client set. The following operations are provided:

- $get(i)$ - returns the bit at index i.
- $insert(i)$ - sets the bit at index i to 1.
- $remove(i)$ - sets the bit at index i to 0.
- $choice$ - returns an arbitrary bit position that is set to 1.

The choice dictionary was not implemented by this author. The library for space-efficient algorithms [Tec18] already contained an implementation for the choice-dictionary data-structure, which was used. Its operations use $O(1)$ time. The space used is $O(n)$ bits.

3.6 Dyck-matching Structure

As a reminder, a dyck word is a sequence of properly nested parentheses which can be stored as a bit sequence. The dyck-matching structure described in this section is based on [GRRR04] and [MR01]. A dyck-matching structure is a structure initialized with a dyck word d which allows us to find the $match_d(p) = p'$ of any parenthesis p in d. The simplest method of executing $match_d(p)$ is by using the Algorithm 3.1, to which we refer as *natural dyck-matching*.

Algorithm 3.1: Natural dyck-matching for finding all matching parentheses in a dyck word

```
Stack S;
for j ← 1 to n do
    if b_j = "(" then
        S.push(j);
    end
    else
        // b_j = "("
        j ← S.pop;
        Output (i, j); // i and j are indices of matching parantheses
    end
end
```

Storing the results of the natural dyck-matching algorithm in a table would give us a matching structure which lets us evaluate $match_d(p)$ in constant time. Initialization takes $O(n)$ time and the resulting table takes $O(n \log n)$ bits space. We call this structure *natural dyck-matching* structure. To achieve a more space-efficient solution, while keeping the constant access time for $match_d(p)$, we start by segmentizing the dyck word $d = (b_1, \ldots, b_n)$ into segments of length $\ell = \lfloor (\log n)/2 \rfloor$, or $\ell = \lfloor (\log n)/2 \rfloor - 1$ if $\lfloor (\log n)/2 \rfloor$ is even. The last segment can be smaller. This segmentizes d into $q = \lceil n/\ell \rceil$ segments, which we call $blocks$. An example of a segmentized dyck-word can be seen in Figure 3.5.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
(	(	)	(	(	)	(	(	(	(	)	(	)	)	)	)	)	)

Figure 3.5: An example of segmentizing a dyck word into blocks of size 5.

Our implementation uses a single value for $\ell = 7$. Similar to our approach for rank-select (Section 3.4) we will employ a table lookup for the blocks. To read blocks of size

$\ell = 7$ we need to employ bit shift operations, since the internal storage type of the bitset is 8, 16, 32 or 64 bits large. This adds a very small constant overhead in accessing blocks for table lookup. We chose 7 as the fixed length because most of our bitset structures used for rank-select internally use 8 bits as the storage type. For a bitset using 8 bit blocks, reading 7 consecutive bits in the bitset at most requires to read two blocks.

We call a parenthesis b *local* if its match b' belongs to the same block as b, and *global* otherwise. Because ℓ is odd, every block—except possibly the last one—has at least one global parenthesis. We define a set P which contains all parentheses that fulfill one (or more) of the following three requirements: (A) from every block that contains at least one global opening parenthesis, the leftmost such parenthesis; (B) from every block that contains at least one global closing parenthesis, the rightmost such parenthesis; (C) the match b' of every parenthesis that belongs to P according to (A) or (B). We call P the set of $pioneers$. We can observe that every pioneer of d is a global parenthesis. With $P = p_1, \ldots, p_m$ where $1 \leq p_1 < \cdots < p_m \leq n$ let d_p be the subword of d that contains only the pioneers, in the same relative order as in d. Then d_p is a proper dyck word itself and $match_{dp}(i) = j$ if and only if $match_d(p_i) = p_j$. In the following descriptions we only specify finding the match of opening parentheses, as finding the match to a closing parenthesis can be done in symmetry.

Let b be a global opening parenthesis in d. Then there is an opening pioneer to the left of or equal to b, and if p is the rightmost such pioneer, then p and b belong to the same block G_b, and their matches also belong to the same block (which is strictly to the right of G_b). The reason for this characteristic is that, since every block with a global opening parenthesis contributes the leftmost such parenthesis to P, p must exist and belong to the same block as b.

Let G'_b be the block that contains the match of b and suppose that the match of p belongs to a different block G'_p and that b is not a pioneer. Because matching pairs of parentheses are always properly nested by definition, G'_b has to be to the left of G'_p; but then the match of the rightmost global closing parenthesis in G'_b is a pioneer that is strictly between p and b. This would contradict the choice of p being the leftmost global opening parenthesis in its block.

We now describe a space-efficient dyck-matching structure D_d as implemented for a word d that allows us to evaluate $match_D$ in constant time. The first step is to create lookup tables for all blocks of size ℓ. Just as our lookup tables we create for rank-select (Section 3.4), every lookup table only exists once, no matter how many dyck-matching structures are created during runtime. Thus, the size of the tables is negligible. The information that we store in the tables is the following: if the parenthesis has a local or a global match, if the parenthesis is a pioneer via rules (A) or (B) and the nesting depth inside the block. To calculate the nesting depth for a parenthesis b_i in $d_b = (b_1, \ldots, b_\ell)$, we assign the function $\rho(b_i) = -1$ if b_i is an opening parenthesis and $\rho(b_i) = 1$ otherwise. Inside a block $D_b = (b_1, \ldots, b_\ell)$ the nesting depth of b_i is $\sum_{j+1}^{\ell} \rho(b_j)$. The next step is to create the dyck word d_p over the set of pioneers P of d. Pioneers that belong to P via rules (A) and (B) can be found with table lookup. For the pioneers that belong to P via rule (C) we need to find the matches of all pioneers that are in P via rules (A) and (B). This is done with natural dyck-matching. For the word d_P we also create a recursive dyck-matching structure D_p with a maximum recursive depth of two (the pioneers of the pioneers), for a visual example refer to Figure 3.7. If

we would exceed two recursions, we instead construct a simple table structure for d_P. Next we create a rank-select structure R_P for the bitset representation of the set P in w. This is done by initializing a rank-select structure with the bitset $C = (c1, \ldots, c_n)$ where $c_i = 1$ if and only if b_i is a pioneer, for $i = 1, \ldots, n$. This allows us to translate between D_d and D_p. Now, given the index $i \in \{1, \ldots, n\}$ of an opening parenthesis $b = b_i$ in d with a dyck-matching structure D_d initialized, we proceed to compute the index $match_D(i)$ as follows: let G_b be the block that contains b. First we consult our lookup table on the segment G_b to determine if the match of b is local (see Figure 3.6). If it is, we can find the position of the match of b' of b within G_b: the local index of the match. Now we only have to calculate the global index b'_g of b', which can be easily done by adding the number of blocks that precede G_b multiplied by the size of the blocks.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
(	(	)	(	(	)	(	(	(	(	)	(	)	)	)	)	)	)

Figure 3.6: Finding local matches in a segmentized dyck-word can be directly obtained via table lookup.

If b is global, we use R_P to locate the rightmost pioneer p to the left of or equal to b, which must exist because the leftmost global opening parenthesis in G_b has to be a pioneer and opening parenthesis. Informally, we compute the index of p in both d and d_p. We use

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
(	(	)	(	(	)	(	(	(	(	)	(	)	)	)	)	)	)

Figure 3.7: Pioneers are marked in purple. The dyck word d_P containing only the pioneers P is a dyck word itself and we construct a matching structure D_P for d_P as well. To translate an index i_p in d_P to an index i in d we employ a rank-select structure R_P.

D_p to find the match p' of p and again use R_p to translate the index p' in d_p to the index p' in d. Let $G_{p'}$ be the block of p'. Then the match b' of b belongs to $G_{p'}$ as well. Thus b' is the leftmost parenthesis in $G_{p'}$ whose nesting depth equals that of b with b' being to the left of or equals p'. In order to identify b', we use table lookup on the block G_p to compute d as the nesting depth of b minus that of p and another table lookup, now on $G_{p'}$ to obtain b' as the leftmost parenthesis in $G_{p'}$ whose nesting depth exceeds that of p and p' by exactly d. Again, this gives us the local index of b' in $G_{p'}$ and we only have to translate it to the global index by adding the number of blocks that precede G_b multiplied by the size of the

blocks.
The dyck word d uses $O(n)$ bits space. The rank-select structure R_P can be constructed in $O(n + (n/r)w)$ bits space, with r equal to the length of the blocks used by the rank-select structure, as described in Section 3.4. Since every block contains at most four pioneers, the recursively built dyck-matching structure D_P uses $O(n/l)$ bits space for each recursion, with ℓ being the length of a block. After two recursions we are left with a sequence of $O(n/(\ell)^2)$ parentheses. At that point we construct a table-based matching structure for the pioneers, which needs $O(n/(\ell)^2 w)$ bits space, with the optimal case of $\ell^2 = \Theta(\log n) = \Theta(w)$ resulting in $O(n)$ bits of used space. Keeping in mind that $\ell < r$, the whole structure takes up $O(n + (n/r)w)$ bits space, with the optimal case of $r = \Theta(\log n) = \Theta(w)$ resulting on $O(n)$ bits used space. In addition to using the default scheme of two recursions for the construction of D_P, our implementation provides the option for the user to choose a different number of recursions.

3.7 Space-efficient Hierholzer Algorithm

The standard algorithm to find Euler trails is the *Hierholzer algorithm*. It's origins go back to the mathematician *Carl Hierholzer* (1840–1871) [Hie73]. The following description of the Hierholzer algorithm is based on [HKL17]. First we define a few keywords and characteristics that we use for the description of the algorithm. Recall that all our implementations are designed to only work for undirected graphs. We call a vertex *white* if none of its adjacent edges have been traversed. We call a vertex *gray* if some, but not all, of its adjacent edges have been traversed. We call a vertex *black* if all of its adjacent edges have been traversed. Furthermore, we call a vertex *odd* if the number of untraversed adjacent edges is odd or *even* otherwise. Take an undirected graph $G = (V, E)$. While G has vertices that are not black, repeat the following steps iteratively: select a vertex u that is odd if possible. If there are no odd vertices, select a vertex u that is gray if possible. If there are neither odd nor gray vertices available, select a white vertex u. If there are only black vertices available, stop the iteration. We call u our starting vertex since it is the first vertex for our Euler trail. If u is gray and uneven remember a marked arc k_{old} out of u. Now choose an arbitrary unmarked adjacent edge e of u. Mark the arc a out of u and initialize a new Euler trail T and add a to T. The endpoint of a is the next vertex v that we visit. Choose an arbitrary unmarked adjacent edge e of v. If there are no such edges adjacent with v we stop the iteration. Otherwise keep iteratively following the edges, marking the arcs and adding them to T. This process greedily extends T as far as possible (until there are no more untraversed adjacent edges at the current vertex v). T only consists of previously unmarked arcs with the first arc starting at u. If we remembered an arc k_{old} we now need to replace the Euler trail T_{old} that contains k_{old} with a combination of T and T_{old}. We do this by inserting all the arcs of T directly before k_{old} in T_{old} keeping the distinct sequence of arcs in T intact. We now end the iteration. If all vertices are now black, the algorithm terminates. Otherwise we continue from the top by selecting a new starting vertex for a new trail. If G is a (semi) Eulerian graph we have exactly one trail after the algorithm terminates. Otherwise there are multiple trails. While the algorithm can result in different trails being created, it will always

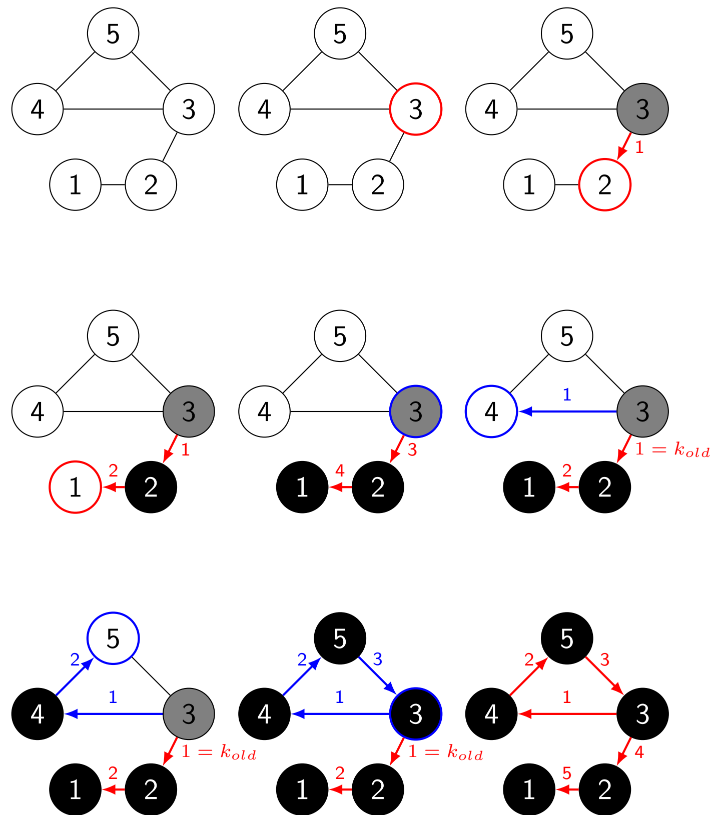

Figure 3.8: Depiction of the steps of the Hierholzer algorithm. We choose vertex 3 as our starting vertex because it is uneven. We start by adding an arbitrary edge adjacent to vertex 3 to the trail. Following and adding edges we arrive at vertex 1, at which point there are no more edges to traverse. We start a new trail, again choosing the vertex 3 as our starting vertex. This time we choose vertex 3 because it is gray (we prefer uneven, then gray, then white). Since vertex 3 is even and gray there is an outgoing, traversed edge adjacent to vertex 3. We remember that edge as k_{old}. After traversing the new trail (blue) to the end, we again arrive at vertex 3. Since we remembered k_{old}, we insert the new trail before k_{old} into the old trail (red). Now that all vertices are black the algorithm terminates.

create an Euler partition. A visual example of the algorithm can be seen in Figure 3.8.

Saving the resulting trails as a list of integers that represent vertices or arcs requires $O(m \log(n+m))$ bits space with the algorithm having a runtime of $O(n+m)$. In order to reduce the space requirements while keeping the same runtime we need to devise a better strategy for storing the resulting Euler partition. To start, we equip each vertex v with a data structure D_v, called its *trail structure*. Take a vertex v of degree $d \geq 1$. The trail structure D_v maintains a partition of $\{1, \dots, d\}$ in three sets we call I, O and U. We also maintain a matching structure M, which manages a matching between a value $i \in I$ and a matching value $o \in O$ (more thoroughly described later). At first, I and O are both empty.

Storing Euler trails now works as follows: each integer k in $\{1, \ldots, d\}$ belongs to O if the kth arc a in the adjacency array of v is traversed, to I if the mate of a is traversed, and to U if neither a nor its mate is traversed. Two integers in $\{1, \ldots, d\}$ are matched if and only if they correspond to edges that are consecutive on a trail and share v as a common endpoint, for which we use the matching structure M to keep track of. More informally, during the construction of an Euler trail, U contains all the arcs that have not been added to an Euler trail yet, I contains all the arcs that were used to enter a vertex, and O contains all the arcs that were traversed while leaving a vertex. Two arcs a, a' are matched when we enter the vertex via the arc a and immediately leave via the arc a'.
Our implementation uses a single bitset to represent $M = (b_1, \ldots, b_d)$. When an arc i is matched, we set the corresponding bit $b_i = 1$ with i in $\{1, \ldots, d\}$. For representing I and O we use a single bitset $D = (b_1, \ldots, b_d)$. When an arc i belongs to I we set the corresponding bit $b_i = 1$ with i in $\{1, \ldots, d\}$ and when an arc i belongs to O we set the corresponding bit $b_i = 0$. This maintains incorrect sets until all arcs of v are traversed, because arcs belonging to U have bits set to 0 or 1 as well (depending on how the bitsets are initialized), but we only need the sets to be valid after traversing all arcs of v, at which point the sets of O, I and M are correctly represented.
We split the operations that are available for D_v in two $phases$. In both phases D_v supports *simple queries* that ask whether U contains any elements or whether a given element of $\{1, \ldots, d\}$ has a match or belongs to I, O or U. For the *first phase* D_v supports these two extra operations: the first, which we call $leave$, selects an (arbitrary) element of U, moves it from U to O, and returns it. The second, which we call $enter$, takes $i \in U$ as an argument and moves it from U to I. Then, if $U \neq \emptyset$, we proceed to select an (arbitrary) element $o \in U$, move it from U to O and match i and o. Then we return o; otherwise we return nothing. Once U becomes empty, D_v enters the *second phase*.
In the second phase, additionally to simple queries, D_v supports an operation that returns the element matched to a given matched element in $I \cup O$ and an operation $marry$ that takes as its arguments elements $i \in I$ and $o \in O$ and matches i and o while unmatching any other elements previously matched to i or o. During the runtime of the Hierholzer algorithm, for a given trail structure D_v, the operation $marry$ can only be called at most twice. This results in the matchings created from calls to $marry$ being nearly invariant. For this reason we store the matches of $marry$ in a simple table which uses $4w$ bits space, which we initialize during the first call of $marry$, and do not change the initial matching structures created before a call to $marry$.
Since our definition of $enter$ allows us to choose any arbitrary element o of U we can choose specific values instead of arbitrary ones and still fulfill its requirements. We follow a simple rule for the choice of o: viewing $\{1, \ldots, d\}$ as cyclic, during each call of $enter$ that begins with U containing at least 2 elements, we choose o as the cyclically first element of U that follows i. To support this property, we maintain the elements of U in a doubly-linked cyclic list by storing for each element $i \in U$ the smallest cyclic distances in each direction (*forward* and *backward links*) to another element $i' \in U$. Note that all the distances in the entire list structure sum to d for each of the two links. Thus, they can be stored (theoretically) in a total of $O(d)$ bits. An ideal implementation would use so-called *self-delimiting numeric values* for the forward and backward links, by preceding (for the forward links) or following

(for the backward links) each binary representation by a unary representation of its length, as described in [HKL17]. Such a representation can then be decoded with the help of table lookup. This way, the cost of storage would remain $O(d)$. At the current point the implementation present in the library for space-efficient algorithms uses simple integer values with a word length of 8, 16 or 32 bits (depending on the size of d) as storage instead of self-delimiting numbers.
With the rule for our choice of o of U, we can observe an important attribute. Once D_v has entered its second phase, if we start iterating cyclically over the bitset D, starting at the index s being the index after the last element to leave U, we can construct a valid dyck word W. For simplicities sake, we now assume that every edge of v is matched. An explanation for the more complex case of v having unmatched edges can be found in [HKL17]. Now, constructing the dyck word W works as follows: starting with an empty word W, for each element d_i in D, if $d_i = 1$, append (to W. If $d_i = 0$ append) to W. We then construct a dyck-matching structure T for W (Section 3.6). This enables us, given a matched element of $I \cup O$, to locate the element of $O \cup I$ matched to it in constant time. A visual example of this is shown in Figure 3.9.

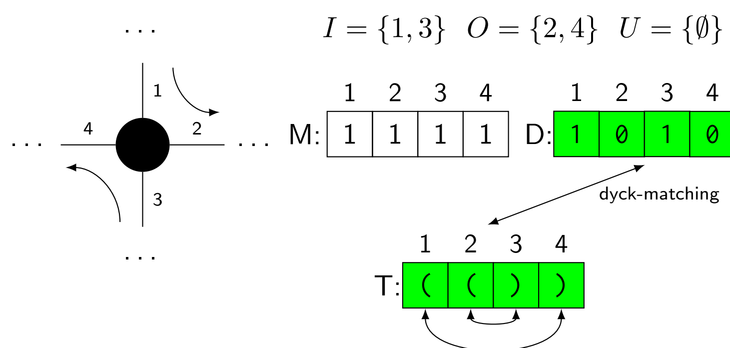

Figure 3.9: Dyck-matching structure that is created for a trail-structure once it enters phase two, with no more elements in U left. The bitset M represents the matched elements of I and O. The bitset $D = (b_1, \ldots, b_d)$ represents $I \cup O$, with $b_i = 1$ if i is contained in I, and $b_i = 0$ if i is contained in O for $1 \leq i \leq d$. The dyck-matching structure T is constructed to find matching elements, with opening parentheses representing elements from I, and closing parentheses representing elements from O. E.g., finding the matching arc to the element 3, contained in I, can be done by evaluating $match_D(3) = match_T(3) = 2$.

With these structures in place we can execute the Hierholzer algorithm as follows: first, select a start vertex u. If u is gray and even, we remember the index of an outgoing arc in the adjacency array of u which we call k_{old}. To initialize T with its first arc, we execute the operation $leave$ on D_u and remember the return value as k_{first} which represents the first arc of T. Appending new arcs to T, when our current vertex is v and the arc just before v on T is a, can be done by executing the function $enter(k)$ on D_v, with k being the position of the mate of a in the adjacency array of v. If T cannot be extended any further than v,

there will be no return value from our call to $enter(k)$ (the implementation uses a $null$ value to represent the absence of a return value). If a value k_{old} was remembered previously, we remember the last argument of $enter$ as k_{last}, which represents the last arc in T. We then combine the trails T_{old} and T. This can be done by calling the function $marry(k_{last}, k_{old})$ on D_v, and, if k_{old} is matched to an element we call k_{pred} (found via the dyck-matching structure of D_v) before the first call of $marry$, we call $marry(k_{pred}, k_{first})$ as well. A visual example of $marry$ and the states of the trail structure D_v during the traversing of v is shown in Figure 3.10. The algorithm ends when all vertices are black. To iterate over the Euler trails, we step through the arcs of the graph and test for each if it is the first arc of an Euler trail. If that is the case, iterate over the arcs of that trail. Take an arc a with a position k in the adjacency array of a vertex u. The arc a is the first arc of an Euler trail, called its *starting arc*, exactly if k has no match in D_v and is an outgoing arc, meaning it belongs to O. Analogous, if the position of the mate of an arc a in the adjacency array of a vertex v is k, a is the *last arc* of an Euler trail if k is unmatched and belongs to I. If a is followed by a' on its Euler trail, the position of a' in the adjacency array of v is equal to the element matched to k in D_v, which can be obtained via the dyck-matching structure T of D_v. Therefore, the Euler partitions can be iterated over in constant time per arc and vertex, resulting in a total runtime of $O(n + m)$. In our implementation we construct a rank-select structure over a bitset $S = (b1, \dots, b_n)$ with $b_v = 1$ for $v \in \{1, \dots, n\}$ if the trail structure D_v has an arc that is the start of an Euler trail, which takes $O(n)$ time to initialize and then allows the output of Euler trails in $O(m)$ time. During the construction of our Euler partition, the cyclical doubly-linked list we use to represent U uses $O(d \log d)$ bits space. Once a trail structure D_v enters its second phase, we destroy the structure U and then create the dyck-matching structure T, which uses $O(d + (d/r)w)$ bits space, with r being the size of the blocks used by the internal rank-select structures (Section 3.4 and 3.6). The bitsets for D and M take up $O(d)$ bits space. This results in a space usage of $O(d+(d/r)w)$ bits for the structure once it enters the second phase. The entire Euler partition then takes up $O(n + \sum_{v=1}^{n} d_v + (d_v/r)w)$ bits space, with d_v being the degree of the vertex v. The optimal case of $r = \Theta(\log d_v) = \Theta(w)$ results in $O(n + \sum_{v=1}^{n} d_v) = O(n + m)$ bits space used.

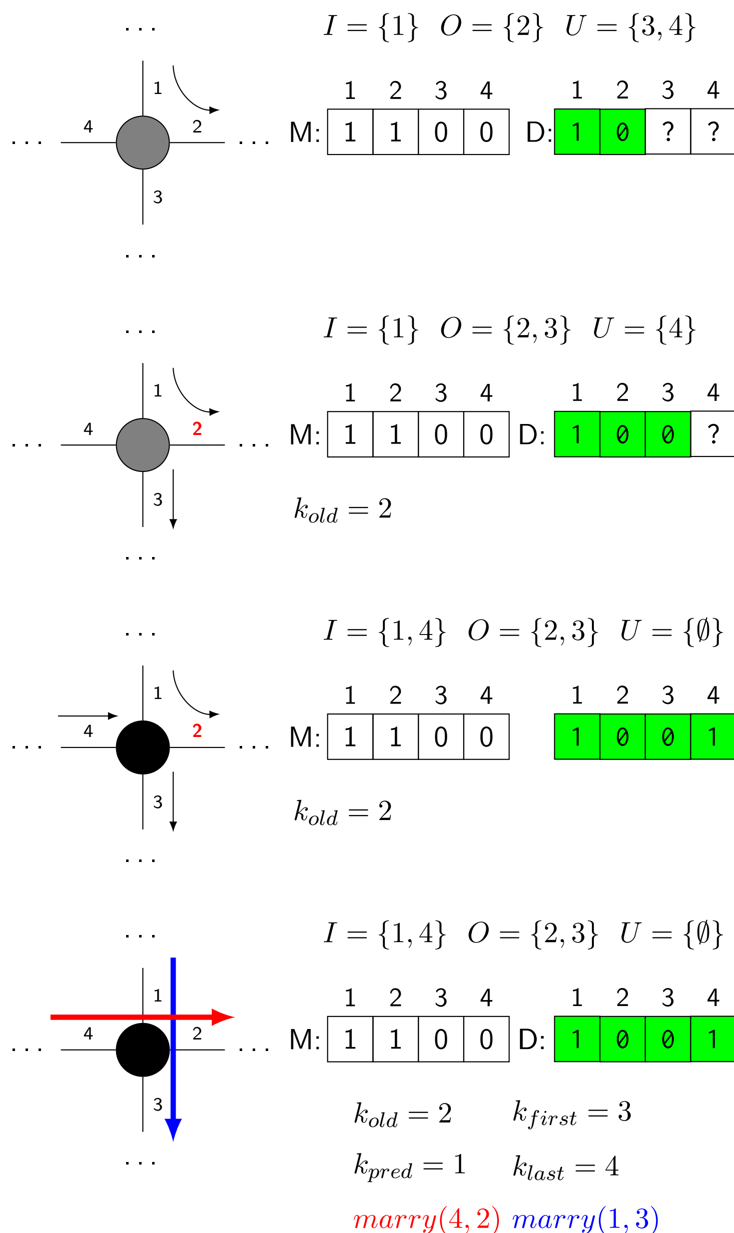

Figure 3.10: Four states of a trail structure D_v for a vertex v with two calls of $marry$. On the left we can see the vertex v and its arcs. The arcs of v are numbered in the order in which they appear in its adjacency array. The arrows represent the path that some Euler trail has taken while traversing v. M is the bitset representing the matched arcs and D is the bitset that represents the set $I \cup O$.

3.8 Subgraph Stack

A *subgraph stack* is a structure that maintains a finite list of ordered, undirected graphs $(G_0, \ldots, G_\ell)$ such that G_i is a subgraph of G_{i-1} with $G_{i-1} \neq G_i$ for $i = 1, ..., \ell$. We call that list the *client list* of the subgraph stack. The subgraph stack is initialized with an undirected graph G_0, which is a graph structure as described in Section 3.3. The user can append a new subgraph $G_{\ell+1}$ of G_ℓ to the end of the list. We call this operation *push*. Removing the graph at the end of the list is called a *pop* operation. The user can call all the regular graph operations on $G_0, ..., G_\ell$ that are provided by the graph interface just as with the regular graph implementation described previously. While from the view of the user the graph G_i is a subgraph of G_{i-1}, internally G_i is not a true subgraph of G_{i-1}, but *isomorphic* to a subgraph of G_{i-1}. The functions to access the isomorphism are accessible by the user. The functions are based on the following characteristics: two undirected graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there are bijections $\phi : V \rightarrow V'$ and $\psi : E \rightarrow E'$ such that for each $e \in E$, if the endpoints of e are u and v, then the endpoints of $\psi(e)$ are $\phi(v)$ and $\phi(u)$. If an edge or vertex is contained in G_{i-1} and G_i we say that it *survived the transition* from G_{i-1} to G_i and G is isomorphic to G' via the functions ψ and ϕ. An example can be seen in Figure 3.11.

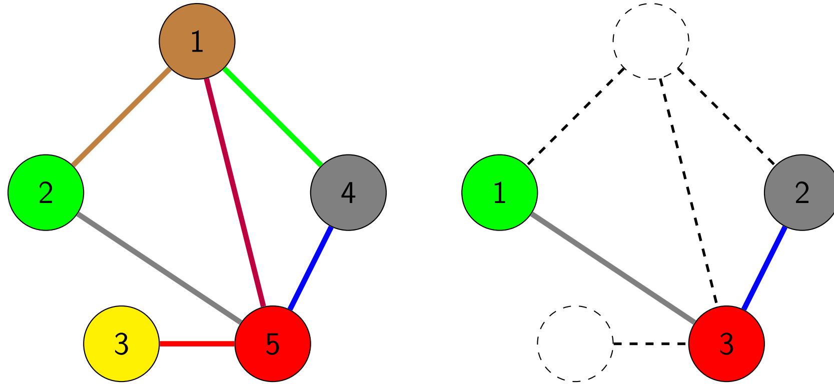

Figure 3.11: Example for isomorphism from G to G'. The colors represent the isomorphism of the edges and vertices. The dashed vertices and edges in G' did not survive the transition from G to G'. I.e., $\phi(1) = 2$, $\phi(2) = 4$, $\psi(3) = 5$.

Besides the regular graph operations we also number the arcs of a graph consecutively starting at 1. This is needed for various other operations described later. Take an undirected graph $G = (V, E)$ with $V = (1, \ldots, n)$ and $m = |E|$. For $j = 1, \ldots, n$ let d_i be the degree of vertex i. We assign the number $g(j, k) = k + \sum_{i=1}^{j-1} d_i$ to the kth arc out of j. We define $g^{-1}(r)$ as the inverse function of $g(j, k)$ as follows: if $g(j, k) = r$ then $g^{-1}(r) = (j, k)$. We call g the *arc-numbering function* of G. Since we want to evaluate g and g^{-1} in constant time, we need to create a data structure to access the values in constant time.
The data structure used for this consists of two rank-select structures. The first one we call P, which is of length $2m$ with 1s in the positions $\sum_{i=1}^{j} d_i$ for $j = 1, ..., d_n$. With P created we can evaluate $g(j, k)$ in constant time with $g(j, k) = select_P(j - 1) + k$ for

$j = 1, ..., d_n$ and $k = 1, ..., d_j$ and for $r = 1, ..., 2m$, $g^{-1}(r) = (rank_P(r-1)+1, r - select_P((rank_P(r-1))$. This works as long as there are no vertices with a degree of zero. To support vertices with a degree of zero we need a second rank-select structure we call Q, whose ith bit is set to 1 if the degree of the ith vertex is not zero. With this in mind the functions $g(j,k)$ and $g^{-1}(r)$ now become $g(j,k) = select_P(rank_Q(j)-1)+k$ for $j = 1, ..., d_n$ and $k = 1, ..., d_j$ and $g^{-1}(r) = (select_Q(rank_P(r-1)+1), r - select_P((rank_P(r-1)))$ for $r = 1, ..., 2m$. Now the arc-numbering function can be evaluated in constant time. Additionally, we can now evaluate the degree of a vertex via the function $d_i = select_P(rank_Q(i) - select_P(rank_Q(i-1)))$. We call P and Q the *arc-numbering arrays* of G. An example can be seen in Figure 3.12. As mentioned, the user can call all the regular graph query functions on each of the graphs in the client list of the subgraph stack. The functions are as follows: $order$, $degree$, $head$, and $mate$ (Section 3.3). With the client list being $(G_0, ..., G_\ell)$ and $0 \leq i \leq j \leq l$ calling these functions for a graph G_i will be written as $order_i$, $degree_i$, $head_i$, and $mate_i$. Additionally, we support the user to call the functions g_i, g_i^{-1} discussed previously. Omitting the parameter i will default to calling the function for G_ℓ. The user can call ϕ_i, ψ_i, ϕ_i^{-1}, ψ_i^{-1} to translate from G_i to G_{i-1} with $\phi_{i,...,j} = \phi_{\ell+1} \cdots \circ \phi_{j-1} \circ \phi_j$ and $\phi_{j,...,i} = \phi_j^{-1} \circ \phi_{j-1}^{-1} \circ \cdots \circ \phi_{i+1}^{-1}$. We define $\psi_{j,...,i}$ analogous.
Now we discuss the $push$ operation. Since we are interested in a space-efficient implementation we do not want to push successive graph objects to the client list. Instead, when the user wants to push a new subgraph $G_{\ell+1}$, the user can call $push(B_v, B_a)$ with B_v being a bit sequence of length n_ℓ and B_A of length $2m_\ell$. These sequences are interpreted as bitset representations of a subset V' of V_ℓ and a subset A' of A_ℓ. What this means is that B_v and B_a contain the information which vertices and arcs from G_ℓ are still contained in the subgraph $G_{\ell+1}$. This is done by simply setting the ith bit in the respective bitset to 1 if the ith vertex or arc is contained in $G_{\ell+1}$. It has to be said that the bits can not be set arbitrary; some requirements have to be satisfied for the bitsets to be valid. For every $a \in A'$ we have to have $g_\ell(mate_\ell(g_\ell^{-1}(a))) \in A'$. Informally, for every surviving arc a', the mate of a' has to survive as well. Reason being that each graph G_i on the stack is undirected. Recall that we represent each edge e in E_i by two arcs, as described in Section 3.3. Another requirement is $head_\ell(g_\ell^{-1}(a)) \in V'$, meaning every endpoint of a surviving arc a' has to survive: an edge cannot be connected to two vertices if those vertices do not exist. Besides these requirements, the bits can be set at any position. A visual example of the created structures can be seen in Figure 3.13.
The implementation adds the option to call $push(B_A)$ which induces the vertices that survive from the surviving arc, i.e. each surviving vertex is an endpoint of the surviving arcs: this results in a subgraph that has no vertices with a degree of zero. Analogous, the user can call $push(B_V)$, which induces the surviving arcs from the surviving vertices.
Defining the vertex set of $G_{\ell+1}$ as $V_{\ell+1} = \{1, ..., |V'|\}$ and its arc set as $A_{\ell+1} = \{1, \ldots, |A'|\}$. Let $\phi_{\ell+1}$ and $\psi_{\ell+1}$ be the bijections from $V_{\ell+1}$ to V' and from $A_{\ell+1}$ to A'. In more detailed terms, these functions map the number of a vertex in $V_{\ell+1}$ or an arc in $A_{\ell+1}$ in G_ℓ to the number that it has in G_ℓ. We also extend the inverse function $\phi_{\ell+1}^{-1}$ of $\phi_{\ell+1}$ as follows: $\phi_{\ell+1}^{-1}(u)$ for $u \in V_\ell$ equals 0 if the vertex u does not exist in $G_\ell + 1$ because it did not survive during the creation of the subgraph. We define $\psi_{\ell+1}^{-1}$ analogous. While this techni-

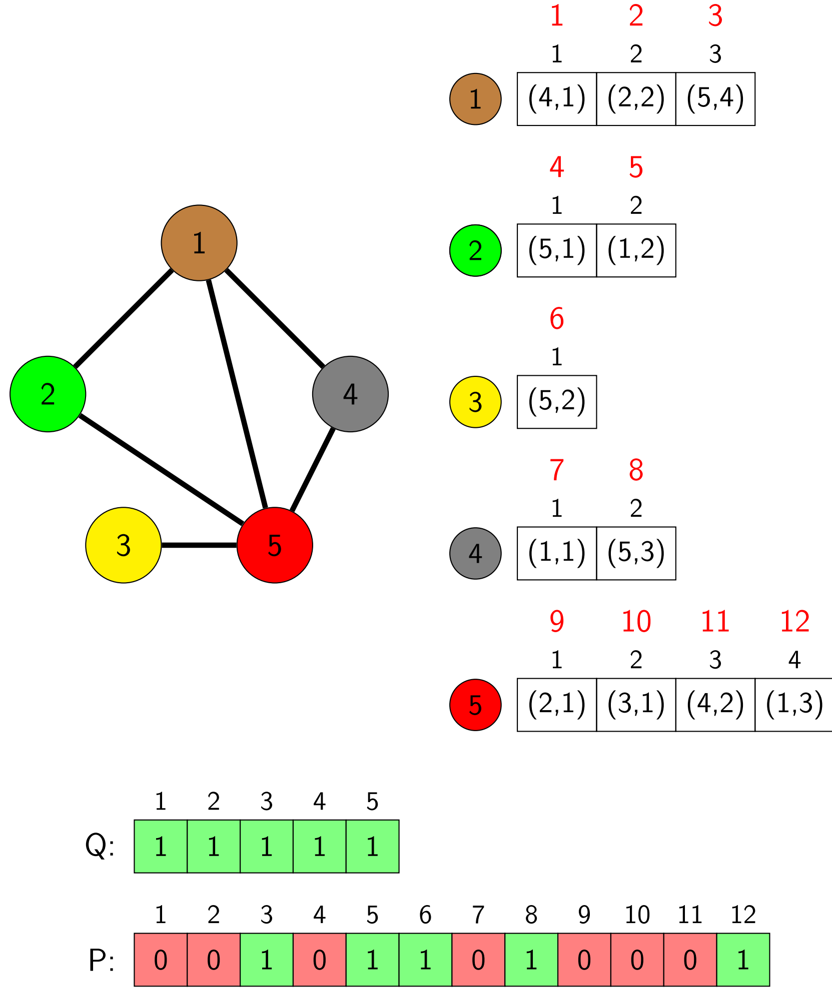

Figure 3.12: The arc-numbering function g for a graph G. The numbers assigned to the arcs can be seen above the indices in the adjacency arrays. Shown at the bottom of the picture is the bitset used for the rank-select structures for P and Q. Since every vertex u in the graph has a degree $d_u > 0$, every bit in Q is set to 1. For P bits are set to 1s in the positions $\sum_{i=1}^{j} d_i$ for $j = 1, ..., d_n$.

cally breaks the rules for inverse functions, this is helpful for later uses and just makes sense intuitively. After creating rank-select structures for these bitsets, we can evaluate $\phi_{\ell+1}$, $\phi_{\ell+1}^{-1}$, $\psi_{\ell+1}$ and $\psi_{\ell+1}^{-1}$ in constant time. Evaluating these functions can be reduced to a simple $select_{B_V}(k)$ for $\phi_{\ell+1}(k)$ and $rank_{B_V}(k)$ for $\phi_{\ell+1}^{-1}(k)$ with $k \in V_{\ell+1}$ and $select_{B_A}(j)$ for $\psi_{\ell+1}(j)$ and $rank_{B_A}(j)$ for $\psi_{\ell+1}^{-1}(j)$ with $j \in A_{\ell+1}$.
The next step is creating the arc-numbering arrays P' and Q' of $G_{\ell+1}$. We can construct P' and Q' in the following way: for $u = 1, ..., n_{\ell+1}$ with $n = |V_{\ell+1}|$, if $d = deg_{\ell+1}(u) > 0$, then append a 1 to Q and append $d - 1$ 0s followed by a single 1 to P'; if $d = 0$ only

append a 0 to Q'. Informally, the degree of a vertex u' in V' of $G_\ell + 1$ is equal to the degree of $u = \phi_{\ell+1}(u')$ subtracted by the number of adjacent edges of u that did not survive the transition from G_ℓ to $G_{\ell+1}$. Initializing P' and Q' this way can be carried out in $O(|B_V| + |B_A|)$. This is how the construction of P' and Q' is sketched in [HKL17]. The concrete algorithm the implementation uses works as shown in Algorithm 3.2.

Algorithm 3.2: Initializing the arc-numbering arrays P' and Q' during a call of $push$.

```
// initialize P' and Q' with 0s
Bitset Q' ← 0
Bitset P' ← 0
degSum_ℓ ← 0
degSum_{ℓ+1} ← 0
// iterate over all vertices in G_ℓ
for u_ℓ ← 1 to order_ℓ do
    u_{ℓ+1} ← φ^{-1}_{ℓ+1}(uR)
    d_ℓ ← deg_ℓ(u_ℓ)
    // Check if u_ℓ is contained in G_{ℓ+1}
    if u ≠ 0 then
        deg_{ℓ+1} ← 0
        // Iterate over all arcs that belong to u_ℓ in G_ℓ
        for i ← degSum_ℓ to degSum_ℓ + d_ℓ do
            // Check if the arc i is contained in G_{ℓ+1}
            if ψ^{-1}_{ℓ+1}(i) ≠ 0 then
                // increment the degree of u_ℓ
                deg_{ℓ+1} ← deg_{ℓ+1} + 1
            end
        end
        // update the bitsets P' and Q'
        if deg_{ℓ+1} ≠ 0 then
            degSum_{ℓ+1} ← degSum_{ℓ+1} + deg_{ℓ+1}
            Q'[u] ← 1
            P'[degSum_{ℓ+1}] ← 1
        end
    end
    degSum_ℓ ← degSum_ℓ + deg_ℓ
end
```

With rank-select structures in place for P and Q recall that we can evaluate ϕ, ϕ^{-1}, ψ, ψ^{-1}, g, g^{-1}, $order$ and deg in constant time. Next we view the function $head$, which can be evaluated as follows:

$$head_\ell(u,k) = \phi_\ell^{-1}(head_{\ell-1}(g_{\ell-1}^{-1}(\psi_\ell(g_\ell(u,k)))))$$

With a quick glance this function might look intimidating. But breaking it down to its components it is quite simple. We first translate the tuple (u,k) to its corresponding arc number a in G_ℓ via the function g_ℓ. Then we translate a to the corresponding arc number a' in $G_{\ell-1}$ with ψ_ℓ. We then translate a' to the corresponding tuple (u',k') with $g_{\ell-1}^{-1}$. Now a recursive call to $head_{\ell-1}$ with (u',k') takes place until we arrive at G_0 at which point $head_0$ is called, which can be evaluated directly since G_0 is a regular graph structure. Now we only have to translate the result of $head_{\ell-1}$ to the corresponding vertex in G_ℓ via a call of ϕ_ℓ^{-1} and we have our final result. Basically, we translate all the way down, and then up. The function $mate$ can be evaluated in a very similar way:

$$mate_\ell(u,k) = g_\ell^{-1}(\psi_\ell^{-1}(g_{\ell-1}(mate_{\ell-1}(g_{\ell-1}^{-1}(\psi_\ell(g_\ell(u,k)))))))$$

We first translate the tuple (u,k) to its corresponding arc number a in G_ℓ via the function g_ℓ. Then we translate a to the corresponding arc number a' in $G_{\ell-1}$ with ψ_ℓ. We then translate a' to the corresponding tuple (u',k') with $g_{\ell-1}^{-1}$. Now the recursive call $mate_{\ell-1}$ with (u',k') takes place until we arrive at G_0 at which point $mate_0$ is called, which can be evaluated directly. We now translate the result of $mate_{\ell-1}$ to the corresponding arc b' in $G_{\ell-1}$. We then translate b' to the arc b in G_ℓ via ϕ_ℓ^{-1} and translate it to the corresponding tuple, which is our result of $mate_\ell(u,k)$.
Both $head_\ell$ and $mate_\ell$ employ a recursive strategy. Informally, they translate from G_ℓ to $G_{\ell-1}$ to $G_{\ell-2}$ and so forth until they arrive at G_0 and the recursion stops and the respective graph query function ($head$ or $mate$) can be evaluated directly on the graph structure. This results in a runtime of $O(n)$ with $n = |(G_0,\ldots,G_\ell)|$. To improve this to a runtime of $O(a)$ with $O(a) = O(1)$ for all practical applications without changing the space requirements of our structure, we need to employ a different strategy. The reason that these functions have a runtime of $O(n)$ is that ϕ_i and ψ_i can only translate from G_i to G_{i-1}. We now introduce the concept of a *reference graph*. When pushing a new graph $G_{\ell+1}$ on a subgraph stack with a client list $(G_0,\ldots,G_\ell)$ we choose a graph G_r with $r \in \{0,\ldots,\ell\}$ and store the functions $\phi_{\ell+1,\ldots,r}$, $\phi_{r,\ldots,\ell+1}$, $\psi_{\ell+1,\ldots,r}$, $\psi_{r,\ldots,\ell+1}$. Previously we have already done this with $r = \ell$ in the form of the rank-select structures P and Q. This scheme can be seen in the Figure 3.14, second from the left. Another possible choice would be to use G_0 as the reference graph for each new graph $G_{\ell+1}$ that we push on the stack as shown in Figure 3.14 on the left side. This would result in constant time for the translation functions: we can translate from any graph G_i that is currently on the stack to G_0 in a single step. Translating from G_0 to another graph G_i can also be done in one step. For example: $\phi_{i,\ldots,j} = \phi_{i,\ldots,0} \circ \phi_{0,\ldots,j}$. The functions $head$ and $mate$ would work in a similar way by replacing the recursive calls with direct translations to and from G_0. The problem with this approach is that each subgraph

G_i on the stack would use the same amount of space for P and Q. With a growing stack this wastes a significant amount of space, since fewer and fewer bits will be set in the structures. What we need is a hybrid approach that gives us effectively constant access-time for all practical purposes while saving space. To achieve this we choose the reference graphs at specific intervals based on a function we call $\log^* n$. It is defined as follows in [HKL17]: for $j \in \mathbb{N}$ we use the subscript ^{j} to denote j-fold repeated function application, i.e., when $f : X \to Y$ is a function, $f^{(1)} = f$ and, for each $j \in \mathbb{N}$, $f^{(j+1)}(x) = f(f^{(j)}(x))$ for all $x \in (f^{(j)})^{-1}(X)$. For all $n \geq 2$, $\log^*(n) = min\{j \in \mathbb{N} | \log^j n \leq 1\}$, we define $\log^*(0) = 1$ and $\log^*(1) = 1$. This function is extremely fast growing: $\log^*(1) = 1$, $\log^*(2) = 2$, $\log^*(3) = 4$, $\log^*(4) = 16$, $\log^*(5) = 65536$ and $\log^*(6) = 2^{65536}$; a number we will not be able to achieve for practical purposes. What this means is that we choose G_0, G_1, G_3, G_{15}, G_{65536} as our reference graphs; this scheme can be seen in Figure 3.14 on the right side. An example of the slowest case for this scheme would be: let $65536 < i < 2^{65536}$, then $\phi_{i,\ldots,0} = \phi_{i,65536} \circ \phi_{65536,15} \circ \phi_{15,3} \circ \phi_{3,1} \circ \phi_{1,0}$. As one can see this gives us a slowdown by factor of at most five. Translating from G_i to G_j with $0 < j < i \leq \ell$ with G_j not being a reference graph would mean we translate from G_i to G_{ri} (the reference graph of G_i) and then from G_{ri} directly to $G_{ri'}$ (the reference graph of G_{ri}) and so forth, until we arrive at G_{rj}, the reference graph of G_j. From G_{rj} we can translate in one step to G_j. Again, this can be done with at most five steps. A more thorough explanation and proof can be found in [HKL17].

Choosing a reference graph when pushing a new graph $G_{\ell+1}$ onto the stack is quite simple; we either choose G_ℓ or the reference graph $G_{r\ell}$ of G_ℓ. Anytime we push a new graph $G_{\ell+1}$, and the size of the client list is at one of the values of our $\log^* n$ function, we choose G_ℓ as the new reference graph. There is one small problem that $push(B_V, B_A)$ has to take care of in the case that G_ℓ is not our reference graph. The bitsets B_V and B_A are in relation to G_ℓ and have to be translated to bitsets B'_V and B'_A that relate to G_r. First we initialize a new bitset each for B'_V and B'_A with initial values of 0 for all bits in B'_V and B'_A. Afterwards, translating B_V to B'_V and B_A to B'_A can be done in $O(|B_V| + |B_A|)$ time. For the case that the user wants to call a large amount of graph query functions for a specific graph on the stack the interface described in [HKL17] mentions a function $toptune$ which speeds up the calls for the current top graph on the stack G_ℓ. The implementation expands this function to work for any graph G_i in $(G_1, \ldots, G_\ell)$. The expanded function $tune_i$ speeds up the calls of the graph query functions for the graph G_i. Omitting the parameter i is equivalent to calling $tune_\ell$. A call of $tune_i$ creates rank-select structures to directly translate from G_i to G_0 and from G_i to G_{i-1} without the added slowdown of the reference graphs. This concludes the implementation of the space-efficient subgraph stack. We achieved $O(a) = O(1)$ runtime for practical applications of all graph query functions while staying space-efficient.

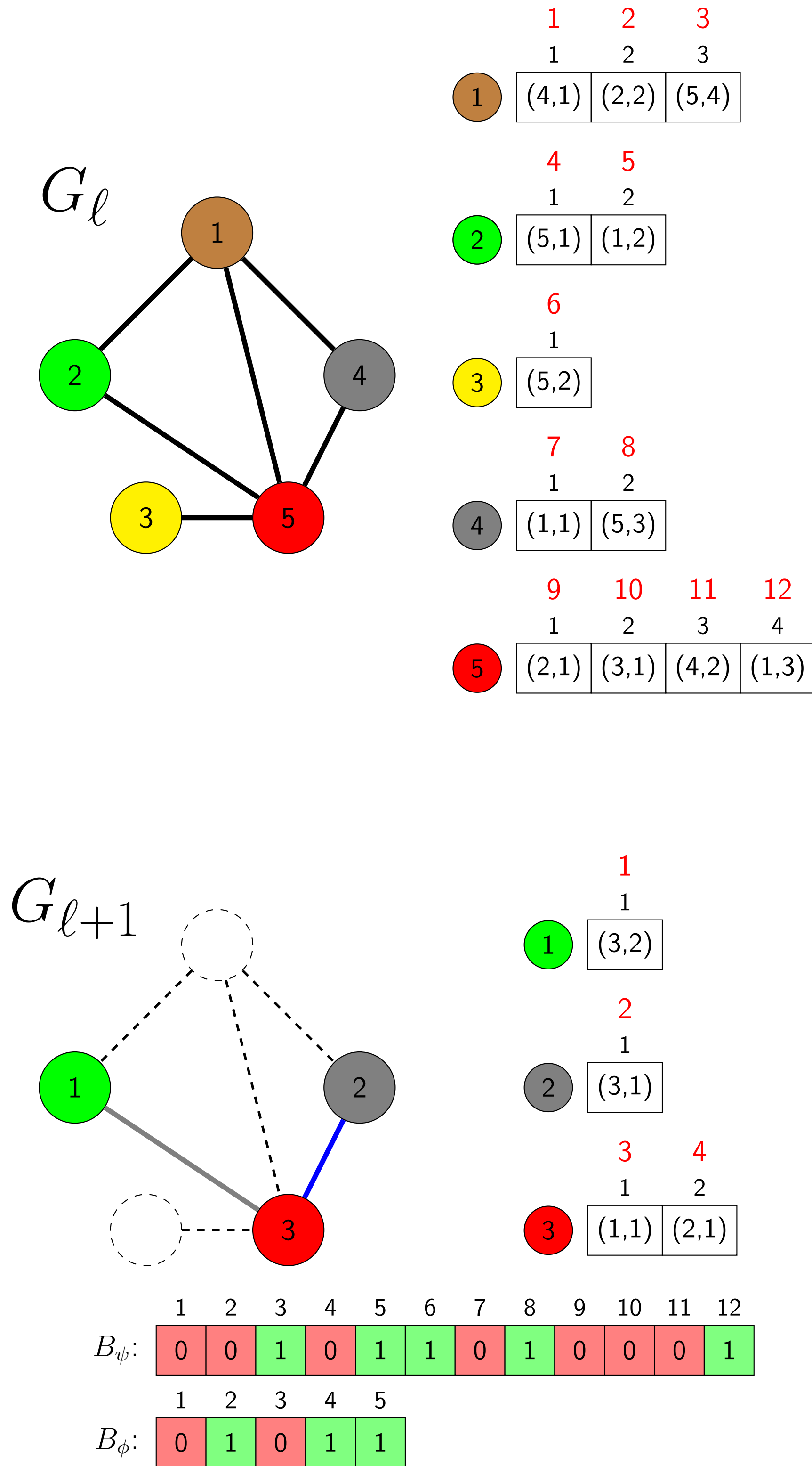

Figure 3.13: The bitsets created for the functions ϕ and ψ when pushing a graph $G_{\ell+1}$. Bits are set when an arc or vertex survives the transition. Rank-select structures created for these bitsets allow the translation between $G_{\ell+1}$ and G_ℓ. While this figure shows adjacency arrays for the graphs, internally it is not represented as such. The values are accessed via functions that use rank-select structures created for P, Q, B_V and B_A.

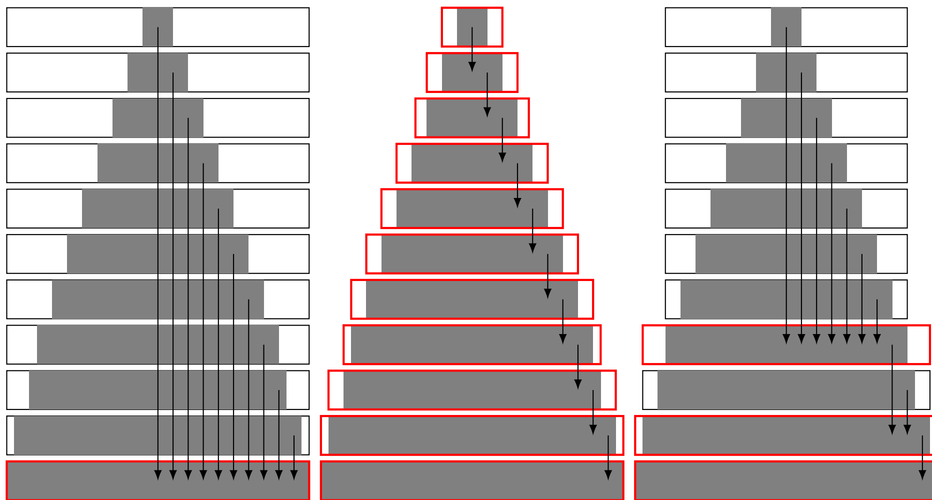

Figure 3.14: Three different schemes for choosing a reference graph and their resulting structure. Each graph is represented by a white rectangle with a gray filling. In detail, the white rectangle represents the size of the rank-select structures for ϕ and ψ and the gray filling represents the amount of set bits in those structures. Graphs that are marked red are reference graphs. The graph at the bottom is G_0 and the graph at the top is G_ℓ.
From left to right:
(1) Choosing only G_0 as a reference graph. Each subgraph uses the same amount of space.
(2) Always choosing G_ℓ as a reference graph when pushing a new subgraph $G_{\ell+1}$. This uses the least amount of space, but has the highest slowdown.
(3) Choosing the reference graph based on the $\log^*(n)$ function. This results in effectively constant access times while saving space.

Chapter 4

Analysis

In this chapter we evaluate our implementation. We start by describing the methodology and results of our runtime and space usage tests, followed by an analysis.

4.1 Methodology

In this section we describe the methodology of the tests we undertook. This includes brief overview of the used hard- and software.

4.1.1 Setting

All our testing has been done on a system with the specifications as follows:

Hardware	
Model	LENOVO L440
Memory	Samsung® SODIMM DDR3 Synchronous 1600 MHz 3.8 GiB
Processor	Intel® Core™i5-4200M CPU @ 2.50 GHz x 4
Graphics	Intel® HD Graphics 4600 Haswell Mobile
OS	Ubuntu 17.10 64-bit
Disk	KINGSTON® SA400S3 117.6 GiB
Swap	6.0 GiB swap partition

The source code for all of our implementations is available in the library for space-efficient algorithms [Tec18]. Everything is written in C++ using the language version C++11 and compiled using gcc 7.2.0 [GNU18].

4.1.2 Measuring runtime and space usage

To measure runtime there are multiple strategies available. One possibility would be to measure the runtime of the entire program using an external clock. This strategy is not suited

for our goal, as we often only want to measure the runtime of certain parts of our program. For example, we want to measure the runtime of some access function on a data structure, but we do not want to include the time it takes to initialize that structure. There are also third-party tools available for profiling the runtime of a program, for example *valgrind* [NS07]. Valgrind offers the possibility to measure the number of CPU instructions that a program has. This can be measured with the tool *callgrind* provided by valgrind [NS07, WKT04]. This is unsuited for our purpose, because we want to measure the runtime, not the number of CPU instructions. Generally, the tools provided by valgrind create a large overhead in runtime as well, which can make using them for profiling multiple algorithms and data-structures very time consuming. Instead, we measure the runtime as follows: we capture the current value of the system clock before the start of some algorithm or block of code as our initial value s. After the algorithm or block of code finishes, we capture the current value of the system clock as our end value e. This results in the runtime $r = e - s$. The current value of the system clock is captured via the function `std::chrono::high_resolution_clock::now`[1] provided by the standard library for C++. The runtime r is converted to milliseconds (ms) after the arithmetic operations. The system clock itself is measured in *clock ticks*, which is the smallest unit of time the system offers. This results in very accurate measurements. Since we can measure any part of a given program this way, it is also very flexible. To measure space, there are a few options as well. One such option would be to read the current value of working memory used by the entire program. In our measurements we want to know the size of specific structures, and generally are not interested in the amount of working memory used by the entire program. For that reason we chose not to employ such a strategy. Another typical approach would be to use yet another tool provided by valgrind called *massif* [NS07, WKT04]. Massif allows to profile the memory usage of a program very accurately. Even though it measures the space allocated at every step of a program, which is stored as an extensive log file, we chose not to use it. Massif adds a very large amount of overhead in runtime, which makes testing very tedious. It also gathers much more information than we actually need. We only want to know the space usage of certain structures used in a program. Therefore, it is not needed to measure every part individually and simultaneously, but only the parts we are actually interested in. For that reason, we chose to measure the space using a different strategy. We implemented a custom allocator[2] we call *tracking allocator*. Anytime the tracking allocator is used for allocation of space of s_a bytes, the value of s_a is added to a global counter variable *byte counter*. Anytime the tracking allocator is used for de-allocation of space of size s_d, the value of s_d is subtracted from the byte counter. Reading the value of the byte counter during an algorithm or after the construction of structures we can measure the allocated bytes that are needed for storage. Employing the tracking allocator only for the allocation of structures we want to measure, helps us to accurately measure used space, while ignoring other parts of the program. The overhead in runtime is also very minimal.

[1] https://en.cppreference.com/w/cpp/chrono/system_clock/now
[2] https://en.cppreference.com/w/cpp/named_req/Allocator

4.2 Measurements

In this section we present the results of our measurements for the implemented structures. We created measurements for the bitset, rank-select, Hierholzer algorithm and subgraph stack. For measurement purposes, we also created a rough draft for the base of an algorithm for bipartite edge-coloring which combines the Hierholzer algorithm and subgraph stack to recursively create subgraphs based on the created Euler partitions.

4.2.1 Bitset

The goal of the measurement of the bitset was to determine if there are any differences in the choice of the block type used for storage. Recall that the internal storage for the bitset is an array of integers, because it is not possible to allocate a single bit of space (Section 3.2). We created bitsets with 8, 16, 32 and 64 bit length blocks for storage, which are all possible sizes of the blocks able to be allocated on our system. We measured the time it took to initialize the bitset with every bit set to zero, and the space used by the structure. The results can be seen in Figure 4.1. We also measured the time that the basic access operations took for each of the bitsets. We measured the time it took to read a block 100 million times. The same metric was used to measure the time used to read a single bit. We did the measurement for bitsets of sizes 100 million to five billion, increasing the size by 100 million every iteration for 50 iterations.

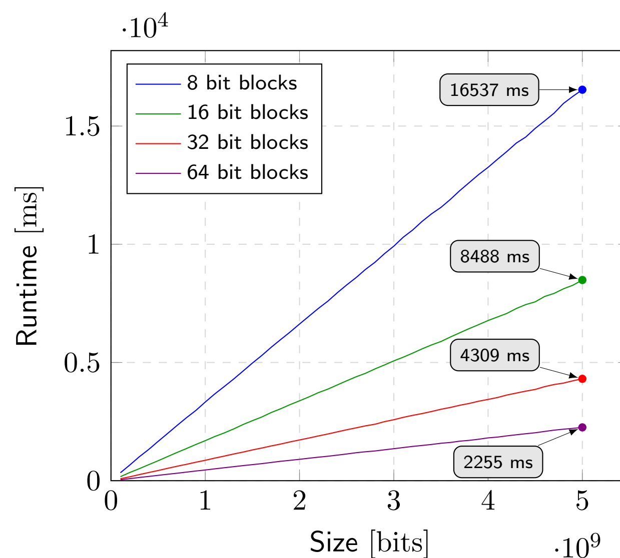

Figure 4.1: Plot showing the runtime of initializing a bitset with different block types.

The access operations showed no difference in runtime between the different block types. The runtime of initializing the bitsets with every bit set to 0 took longer with the smaller block types, because initializing the bitset is done by setting each block to 0 and not each

bit individually. Meaning we can set ℓ bits to 0 with a single operation, with ℓ being the bits contained in a block. We observe this affect the runtime directly by the number of bits per block. The 32 bit version takes double the time as the 64 bit version, the 16 bit version again takes double the time as the 32 bit version and the slowest version at 8 bits takes double the time as the 16 bit version. Most of our structures use a bitset with 8 bit blocks, even though the 8 bit block is the slowest version in initialization. Reason being the use of table-lookup (Section 3.1) by many of our structures. At this point, most of our lookup table structures are initialized for 8 bit blocks, so there is a need to read blocks of exactly 8 bits with a single operation. This is only provided when we use 8 bit blocks for the internal storage of our bitset.

4.2.2 Rank-select

To measure rank-select we chose to implement a comparable rank-select structure that has no aim of being space-efficient, which we call *naive rank-select*. The naive rank-select structure is initialized by computing the values for $rank$ and $select$ in $O(n)$ runtime and storing them in an array using $O(n \log n)$ bits space, similar to the strategy of initializing lookup tables for our space-efficient rank-select structure (Section 3.4). First, we measured the runtime and space requirements of initializing the naive rank-select structure and the space-efficient rank-select structure. To do that we created bitsets with a size of one million bits up to 50 million bits, increasing the size by one million bits per step. We initialized the bitset to have every second bit set to 1. We then measured the time it took to initialize each structure. After initializing, we measured the space occupied by each structure. The size of the initial bitset is not included in the measurement. The results can be seen in Figure 4.2. Secondly, we measured the runtime of calling $rank$ and $select$ by measuring the time it took to call the respective function one million times. Again, we did this for bitsets with a size of one million bits up to 50 million bits, increasing the size by one million bits per step. The results can be seen in Figure 4.3.

The measurements show that initializing the space-efficient implementation of rank-select is not just about ten times smaller, but is also actually faster than the naive implementation. The naive implementation has to initialize much larger arrays as an underlying structure for storage, which takes a lot longer. The naive implementation also has to iterate over each bit individually, while the space-efficient implementation employs table lookups to iterate over multiple bits at once. The functions $select$ and $rank$ take longer for the space-efficient version, which is to be expected. The naive version can be evaluated extremely fast, since only a single access operation to an array has to be computed. Recall from Section 3.4 that the space-efficient version has to calculate local and global values to create the final value of $rank$ and $select$. The fact that $select$ takes longer, is due to the fact that $select$ employs a more complicated call of operations, which includes calls to $rank$ itself. For very large bitsets the initialization time of the naive structure overshadows all gains that could be achieved from the faster access times. Keeping in mind that the values for the runtime of $select$ and $rank$ are for one million function calls. The largest bitset we created measured $5 \cdot 10^9$ bits. The naive implementation took 4398 ms to initialize, while the space-efficient version took only 1419 ms. For the naive version to be as fast in a practical

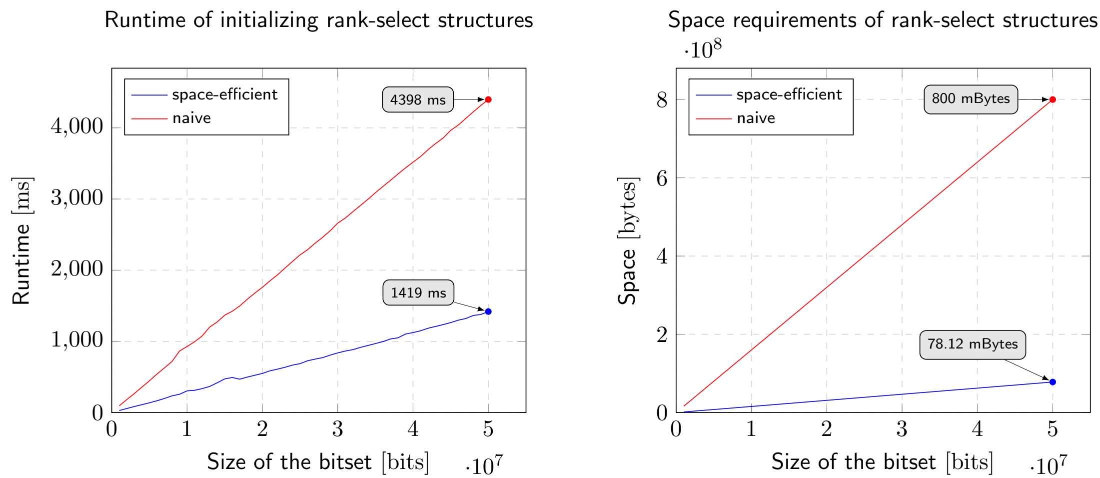

Figure 4.2: Plots showing the runtime and space requirements of initializing rank-select structures. The bitsets used have half of their bits set to 1.

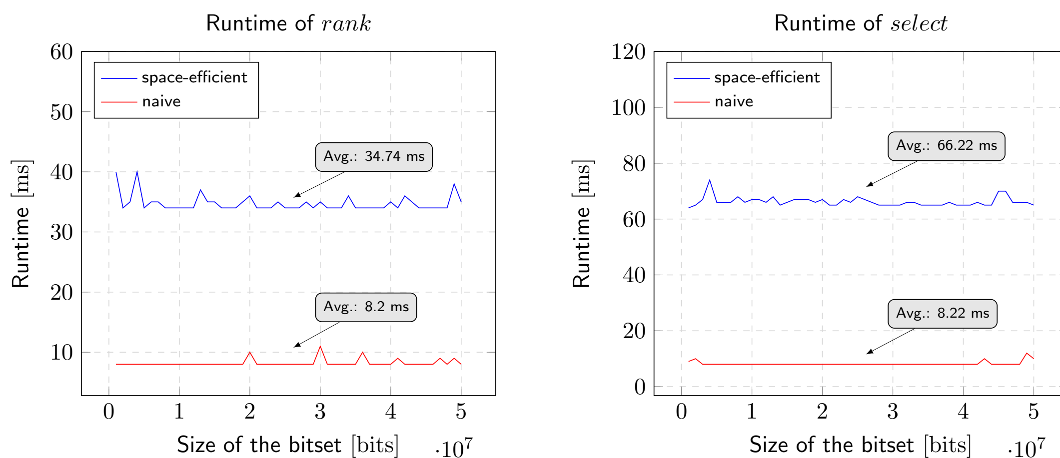

Figure 4.3: Plot showing the runtime of the *rank* and *select* functions. The bitsets used have half of their bits set to 1. The runtime is measured for one million function calls.

situation we would need to call *rank* over one billion times. For very small bitsets with a lot of calls to *rank* and *select* the naive version can outperform the space-efficient version in runtime. Having such a good implementation for the space-efficient rank-select structure is very beneficial for the library of space-efficient algorithms [Tec18]. Rank-select is a very versatile structure, used in many other space-efficient algorithms and structures. Recall that we segmentize the bitset used for rank-select into blocks of a certain size. With the current implementation we only use blocks with a size of 8 bits. Further testing can be done to

determine if other block sizes can deliver better runtimes or result in less space-usage. Since we want to have only one lookup table for any number of rank-select structures, the only other feasible size for blocks would be 16 bits. The next larger possible size would be 32 bits, at which point the lookup table structures become too large for most applications. Our bitset implementation shows faster initialization times for block types (4.2.1) of size 16 bits compared to 8 bits. It is likely that for rank-select structures, for sufficiently large enough bitsets, that a 16 bit based lookup table would give even better results in access times and space usage.

4.2.3 Hierholzer algorithm

To compare the runtime and space requirements of the space-efficient Hierholzer algorithm we implemented a version that we call *naive Hierholzer algorithm*. The algorithm works almost exactly as the space-efficient implementation, but instead of using the trail structure described in Section 3.7, the in- and outgoing arcs for a vertex are stored in arrays. To measure the runtime and space requirements we generate (bipartite) graphs randomly with the following function: create two sets of vertices V_1 and V_2 with $|V_1| = |V_2| = n/2$. Then generate every possible edge with an endpoint in V_1 and the other in V_2 with a chance p. This results in an average value of $p(n/2)^2$ edges per graph. The seed for the random generation has been kept the same for the space-efficient and naive implementation, so that both algorithms always work on the exact same graph. We created random graphs for n-values from 1000 to 60000, increasing by 1000 each step. The value of p is kept constant at 0.1. The results can be seen in Figure 4.4 and Figure 4.5.

In our measurements we can immediately see the effect the overhead of memory management can have on the runtime of an algorithm. Once the operating system has to do extensive swap operations the naive implementation gets throttled heavily. For small domains the naive implementation runs faster by a factor of roughly five, while for larger domains the factor becomes only about 1.3. Once the naive algorithm can't complete the task, the difference in runtime becomes irrelevant: only the space-efficient implementation runs through. The space requirements of both implementations are largely dependent on the amount of edges, which is to be expected, as an Euler partition is just a sequence of edges. The space-efficient implementation takes up less than a quarter of the space of the naive implementation, regardless of the size of the input.

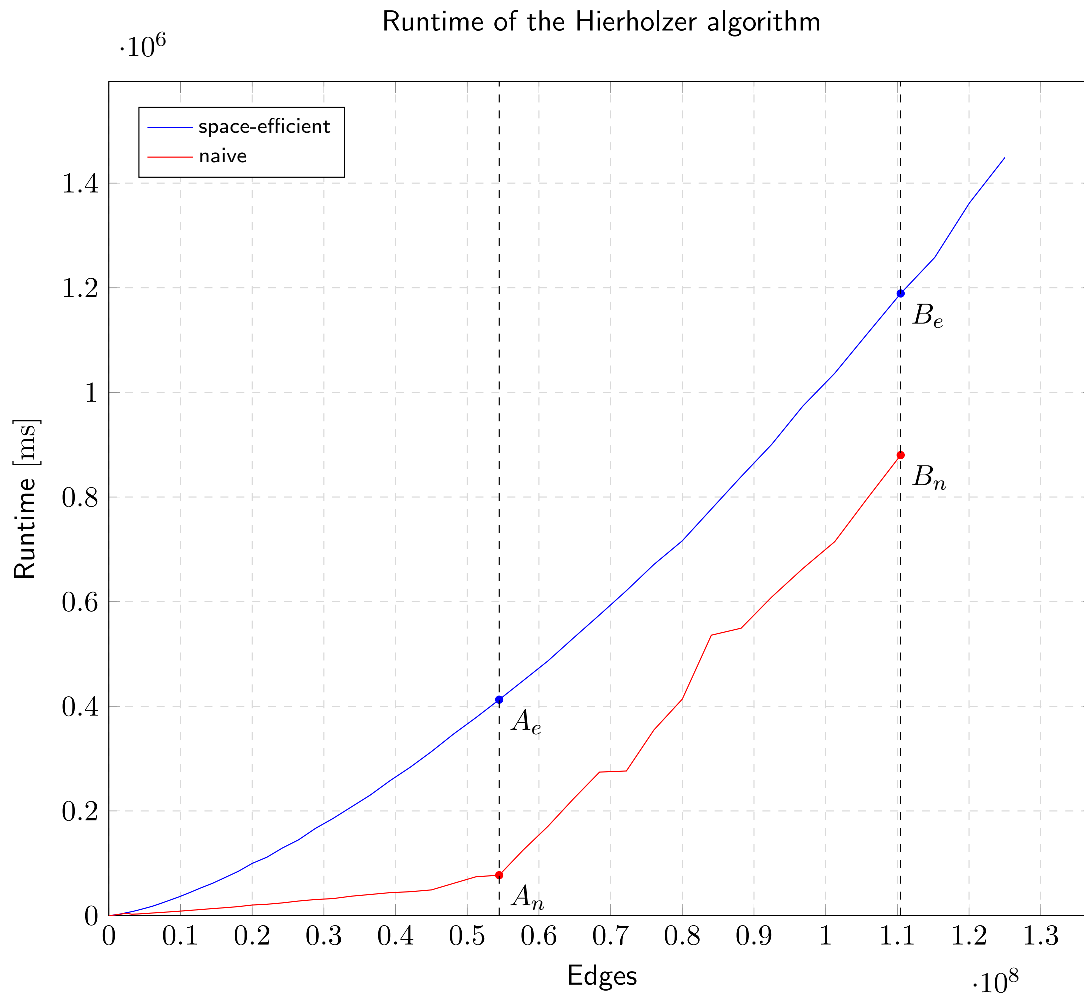

Figure 4.4: Plot showing the runtime of the naive (red) and space-efficient (blue) Hierholzer algorithm. We can see that the time requirements for small domains is much larger for the space-efficient implementation. Starting at point A_n we can see the effects of the memory management overhead, once the operating system needs to make extensive use of the swap partition. The point B_n marks the largest instance that could be run on the system before running out of memory for the naive implementation, which was $n = 47000$. The space-efficient implementation is not affected as much by the overhead of memory management, because there is no need for extensive swap operations. At the points A_e an A_n the runtime of the naive implementation is only 18.7% of the space-efficient implementation, while at the points B_e and A_e it is 74%. At any point after B the space-efficient implementation is superior by default, because it is the only version that was still able to complete the task.

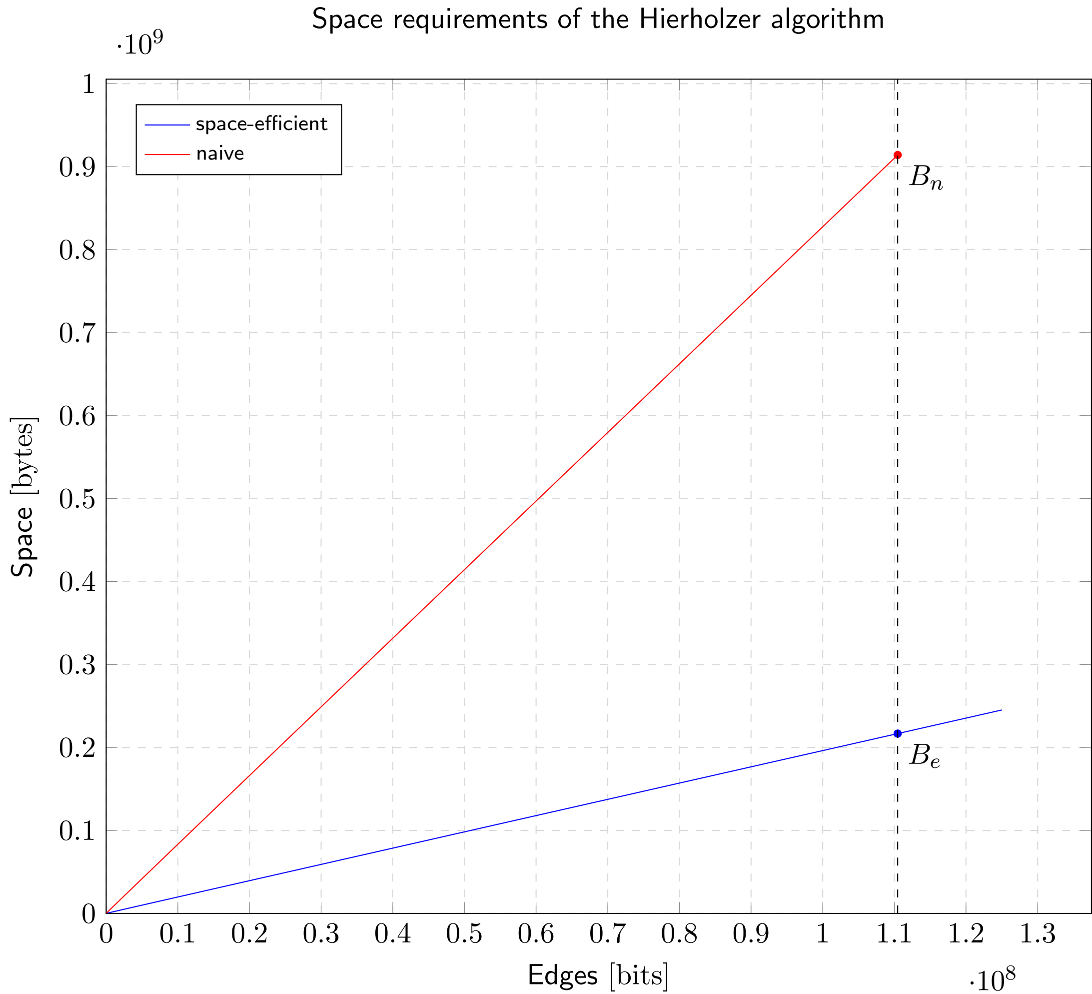

Figure 4.5: Plot showing the space requirements of the naive (red) and space-efficient (blue) Euler partitions created by the Hierholzer algorithm. We can see a linear growth in size for both implementations, with the growth being directly related to the number of edges a graph contains. At the points B_n and B_e the system is running out of memory for the naive implementation, and could not finish the task of creating partitions anymore. The space-efficient implementation still continues to run. On average, the space-efficient implementation took up 23.73% as much space as the naive implementation.

4.2.4 Subgraph stack

To compare the runtime and space requirements of the space-efficient subgraph stack we implemented a version that we call *naive subgraph stack*. When pushing a new subgraph $G_{\ell+1}$ onto the naive subgraph stack, with G_ℓ being the previous graph on the top of the stack, we create a new graph object for $G_{\ell+1}$ as a copy of G_ℓ and mark the edges and vertices that are not included in the subgraph. We then create a mapping between $G_{\ell+1}$ and G_ℓ to translate between the two graphs using integer values. This is a typical strategy of implementing such a structure that would be used by algorithms working on subgraphs. The base graph with which we initialize the subgraph stack structures is generated with the same method as described in Section 4.2.3. We created random graphs for n-values from 1000 to 35000, increasing by 1000 each step. The value of p is kept constant at 0.1. Each new subgraph $G_{\ell+1}$ we push has every eighth edge removed in the transition from G_ℓ to $G_{\ell+1}$. Then, if a vertex v in $G_{\ell+1}$ has a degree of zero, it is removed. We continue to push new subgraphs until the client list of each subgraph stack contains $\ell = 50$ subgraphs, at which point we stop. The results can be seen in Figure 4.6 and Figure 4.7.

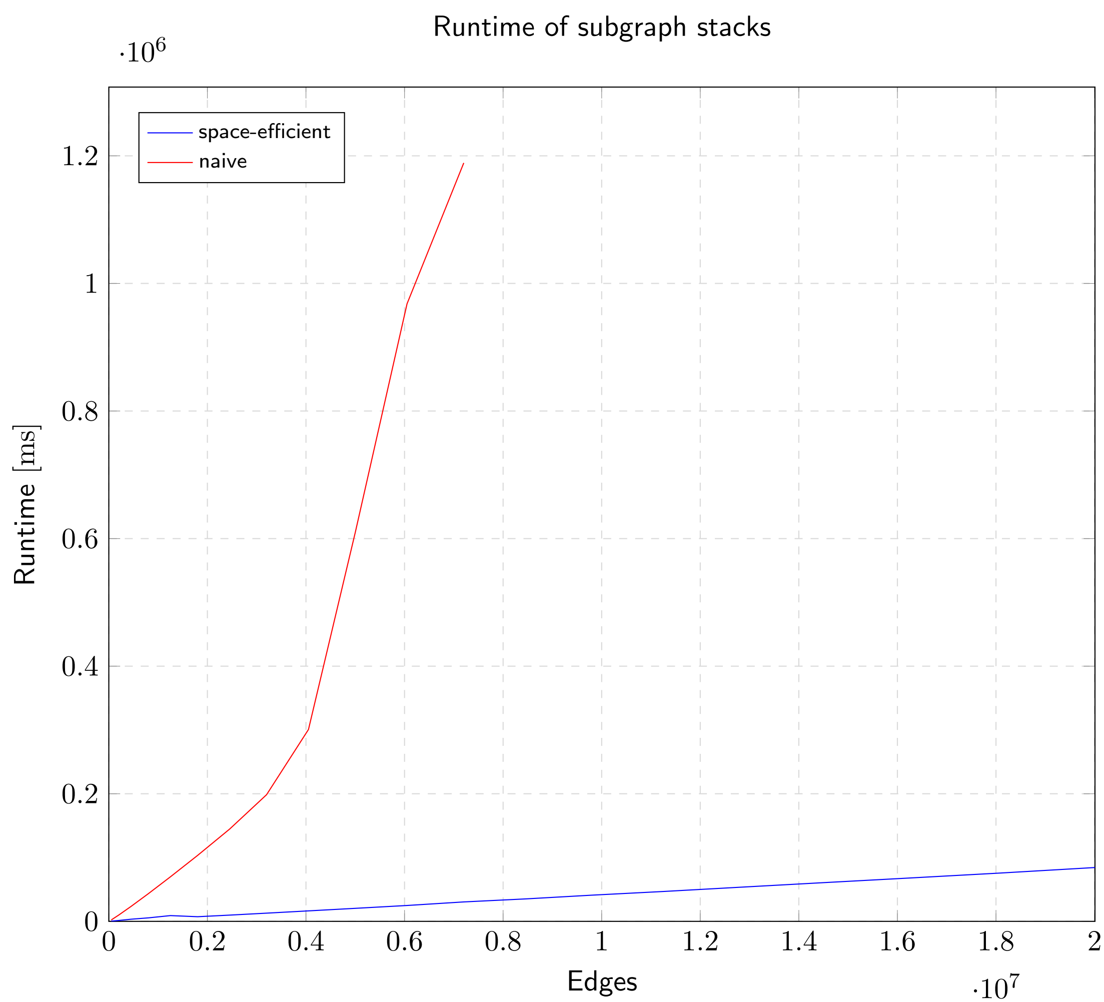

Figure 4.6: Plot showing the runtime of the naive (red) and space-efficient (blue) subgraph stacks. On our hardware the naive implementation does not work for $n > 12000$. The average runtime of the space-efficient implementation for values of $n = 1000$ to $n = 12000$ was less than 3.8% of the naive implementation. The plot is showing the results up to $n = 20000$.

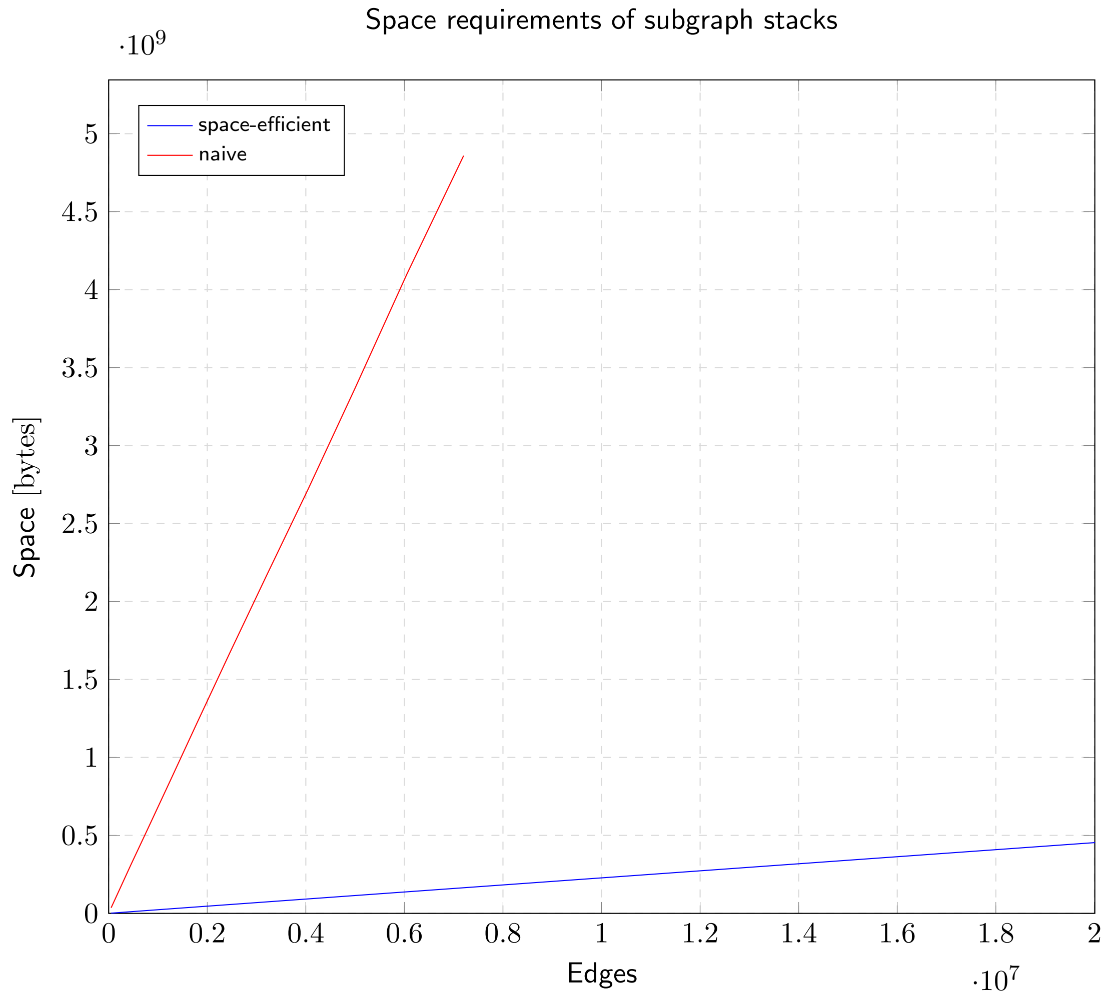

Figure 4.7: Plot showing the space requirements of the naive (red) and space-efficient (blue) subgraph stacks. On our hardware the naive implementation does not work for $n > 12000$. For values of $n = 1000$ to $n = 12000$ the space-efficient implementation uses less than 3.4% of the space of the naive implementation on average. Both implementations show a linear growth in size related to the number of edges. The plot is showing the results up to $n = 20000$.

Our measurements show that the naive implementation is much slower and occupies more space at every tested value of n compared to the space-efficient implementation. The runtime is extremely slow at 26 times that of the space-efficient version on average. The space requirements are sky high as well. The naive implementation does not even run anymore after input graphs with $n = 8000$. This is not surprising: creating new, large graph objects and map structures for every call of push is magnitudes slower and more space consuming than creating simple rank-select structures, which is what the space-efficient implementation uses for the internal structures. It is unlikely that there is a use-case where the space-efficient implementation would be worse than the naive implementation for practical purposes.

4.2.5 Combining the Hierholzer algorithm and subgraph stack structure

To simulate how the space-efficient implementation of the Hierholzer algorithm can be used together with the space-efficient subgraph stack structure we implemented a simple algorithm that works similar as the bipartite edge-coloring algorithm described in Section 2.2. For a given graph G_0 we initialize a space-efficient subgraph stack structure. Then, we create an Euler partition P_0 of G_0 via the naive or space-efficient Hierholzer algorithm. We iterate over the edges of the partition P_0 and create two subgraphs G'_0, G''_0 of G_0, alternately putting edges in the partition in one or the other graph. The created subgraphs now contain roughly half the edges of G_0 each. We push the first new subgraph G'_0 onto the stack and create an Euler partition for it, then stepping through the created trails and again splitting the graph into two subgraphs. We recursively create partitions and push new subgraphs until the size of the subgraph stack is R at which point we stop the recursion. We combined both the space-efficient and naive Hierholzer algorithm with the space-efficient subgraph, because the naive subgraph structure we implemented can be deemed obsolete when viewing our measurements (Section 4.2.4). Nonetheless, to see how the naive subgraph-stack is suited for this case, we also implemented a similar algorithm that uses the naive subgraph stack. Instead of using Euler partitions to create subgraphs, we just create arbitrary subgraphs which contain half the edges of their supergraph. We again do this recursively, and stop once the subgraph stack contains R subgraphs. We created random graphs for n-values from 1000 to 10000, increasing by 1000 each step. The value of p is kept constant at 0.1, and we stop the recursion for $R = 10$. The results can be seen in figures 4.8 and 4.9.

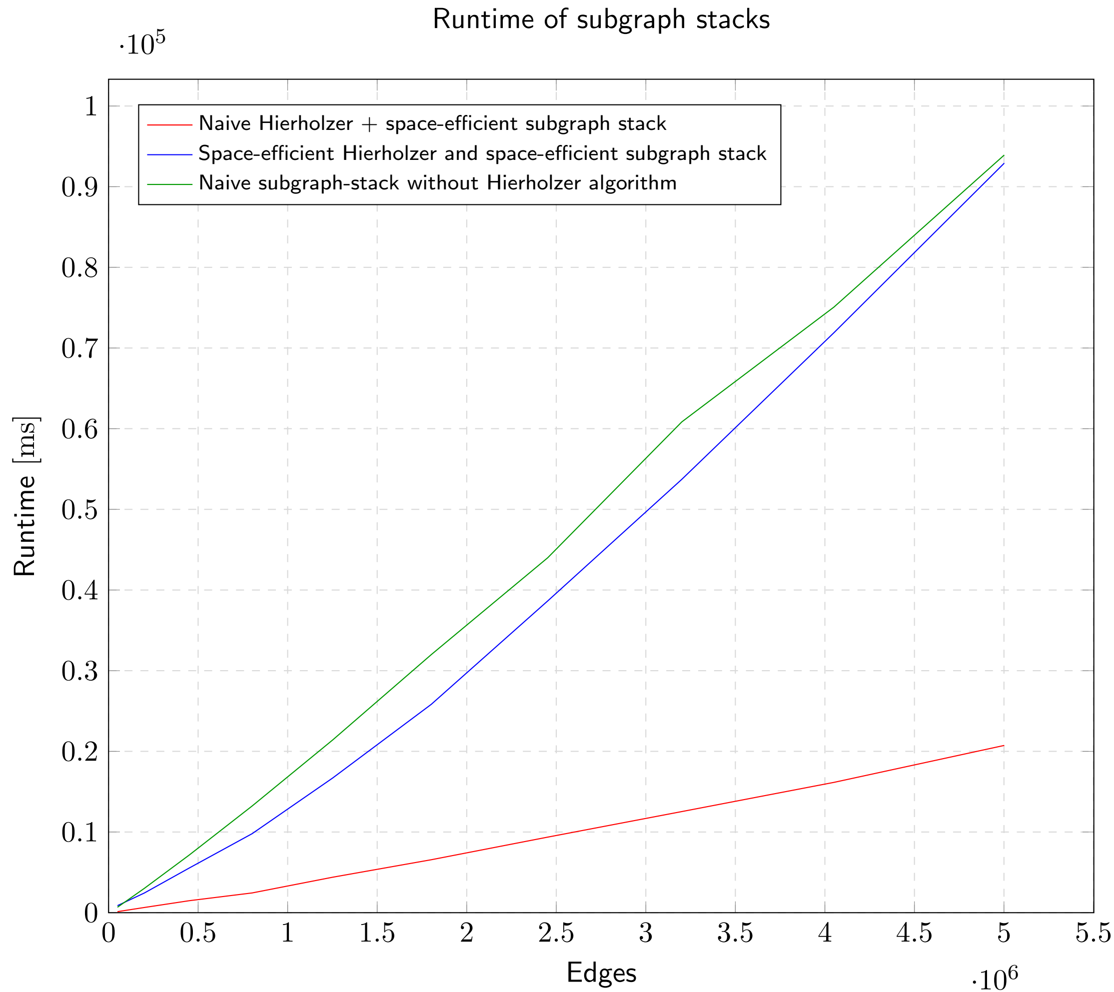

Figure 4.8: Plot showing the runtime of the naive and space-efficient implementation of the algorithm described in Section 4.2.5. Both versions of the Hierholzer algorithm use the space-efficient implementation of the subgraph-stack. The naive subgraph stack chooses arbitrary subgraphs, instead of subgraphs based on Euler partitions. The overhead of creating Euler partitions with the Hierholzer algorithms is large, especially when using the space-efficient implementation. Even with that overhead, the runtime of the space-efficient subgraph stack is lower than the naive subgraph-stack, which does not use Euler partitions for creating subgraphs, but just creates arbitrary subgraphs containing half the edges of their respective supergraph. We obtain a massive runtime gain by using the space-efficient subgraph-stack. Compared to that, the choice of using the space-efficient or naive Hierholzer algorithm only provides marginal gains.

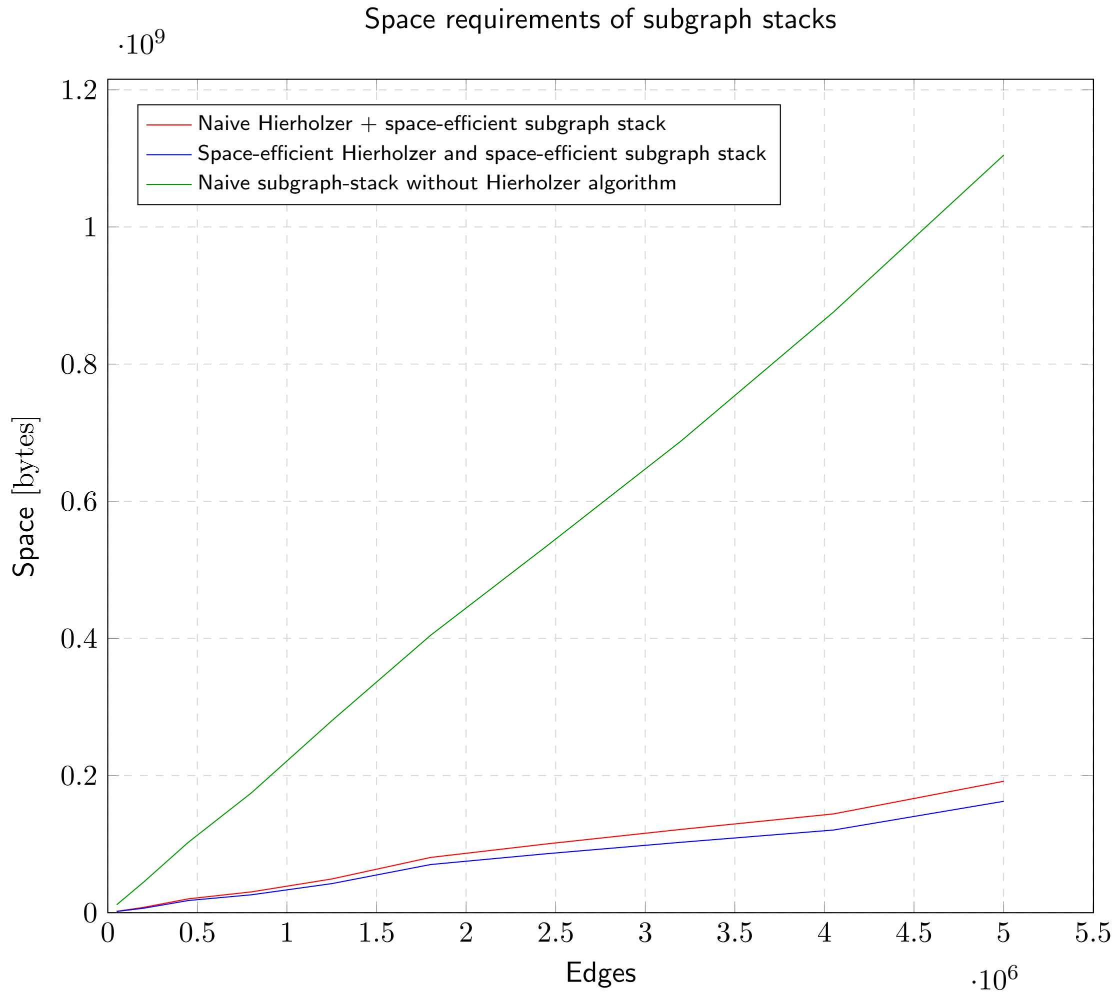

Figure 4.9: Plot showing the space-requirements of the naive and space-efficient implementation of the algorithm described in Section 4.2.5. Both versions of the Hierholzer algorithm use the space-efficient implementation of the subgraph-stack. The naive subgraph stack chooses arbitrary subgraphs, instead of subgraphs based on Euler partitions. The space overhead of creating Euler partitions with the Hierholzer algorithms is large, especially when using the naive implementation. The space-requirements of the space-efficient subgraph stack combined with the space-requirements of the Euler partitions is lower than the naive subgraph-stack, which has no additional space overhead created from choosing the subgraphs. We obtain most of our gain in space-efficiency by using the space-efficient subgraph-stack. By comparison, the choice of using the space-efficient or naive Hierholzer algorithm only marginally affects the space-requirements.

Keeping in mind that in our measurements both versions of the Hierholzer algorithm use the space-efficient implementation of the subgraph-stack, we observe that the naive implementation of the subgraph stack without creating Euler partitions is slower and less space efficient than both combinations of our space-efficient subgraph-stack. The runtime overhead of creating Euler partitions with the Hierholzer algorithms is large, especially when using the space-efficient implementation. Even when using the space-efficient Hierholzer algorithm combined with the space-efficient subgraph stack we have lower runtime than the naive subgraph-stack, which creates arbitrary subgraphs, containing half the edges of their supergraph. This reveals that most of the improvements in runtime comes from choosing the space-efficient subgraph stack. The naive implementation of the subgraph stack requires an enormous amount of space, even without the added space overhead of the Euler partitions. Compared to that, the space-efficient subgraph stack using either of the Hierholzer algorithms uses far less space. This affirms that most gains in space efficiency stem from our choice of subgraph-stack. Compared to that, only marginal gains can be achieved from the use of the space-efficient Hierholzer algorithm over the naive algorithm. We can also see the runtime overhead created by the space-efficient Hierholzer algorithm when comparing it to the naive Hierholzer algorithm. The observed runtime difference is larger than measured in Section 4.2.3. The reason is as follows: in this test we do not just create the Euler partitions, but we also traverse each Euler trail contained in the partition. In the previous measurements we only created the partition. Traversing the Euler partitions created by the space-efficient algorithm has the added overhead of using the dyck-matching structure (Section 3.6). But, even with that large overhead, using the space-efficient Hierholzer implementation and combining it with the space-efficient subgraph stack gives us a much faster runtime than we would have when using the naive subgraph-stack with the naive Hierholzer algorithm. Reason being the extreme overhead of the naive subgraph-stack.

4.3 Conclusion

Our measurements show promising results. In our tests we discovered that the space-efficient rank-select implementation is superior in initialization runtime and space usage compared to the naive implementation. As expected, the runtime of the access operations is slower for the space-efficient implementation. Given the large difference in initialization time, even with the slower access times, the space-efficient rank-select structure runs faster in most applications. Still, there is a possibility this could be improved upon. Recall that the space-efficient rank-select implementation segmentizes the input bitset into blocks of a fixed size, for the use of table lookup. With the current implementation we only use blocks with a size of 8 bits. Further testing can be done to see if other block sizes can deliver better results in space usage and runtime. The only other feasible size for blocks would be 16 bits, which would be the next larger size of the blocks used for internal storage by the bitset structure for which we create our rank-select structure. Our bitset implementation shows faster initialization times for larger block types (4.2.1), and larger block sizes would likely increase the speed of the rank-select initialization as well. Thus, it can be assumed that for rank-select structures created for sufficiently large bitsets, that a 16 bit based lookup table

would offer even better results. The next larger size of blocks used for internal storage for our bitset implementation would be 32 bits. Creating a lookup table structure for such large blocks is often not feasible, as a result of the large space requirements for such a table. For extremely large bitsets this might still be a possibility.
Our measurements of the space-efficient Hierholzer algorithm produced some promising results as well. For sufficiently large inputs it has a very similar runtime to the naive implementation. The biggest use seems to be cases where the naive implementation will not run anymore, because the system does not have enough space capacity. At that point, it does not matter that the space-efficient implementation runs slower than the naive implementation; it is the only version that actually runs through. It is not inconceivable that with further optimizations the use-cases for the space-efficient Hierholzer algorithm could expand. Once the Euler partition is created, most of the space is taken up by the dyck-matching structures created for each vertex of the initial graph (Section 3.6 and Section 3.7). Thus, the main target for improvement would be the dyck-matching structure. Recall that, during initialization of the dyck-matching structure, the input dyck word d is segmentized into blocks of size 7. Similar to the possible improvements proposed for the rank-select structure, a different block size could result in faster runtime and less space-usage. In our measurements of combining the Hierholzer algorithm with the space-efficient subgraph stack we observe the overhead in runtime of traversing the created Euler partition (Figure 4.8). The partition created by the naive Hierholzer algorithm can be read very fast, since they are stored in a simple array. The space-efficient implementation uses the complex operations of the dyck-matching structure, instead of simple array lookup. With that overhead in mind, practical use-cases for the space-efficient implementation might not include iterating over the created partitions. Instead, the user might only have the need to know the number of Euler trails contained in a given Euler partition. Both the naive and space-efficient implementation offer that functionality in constant time after initialization. At that point, the difference in runtime will be the same as shown in Figure 4.4. To use the space-efficient Hierholzer algorithm in a more effective way for recursive graph algorithms, a possible approach would be to combine the gains in space usage of the space-efficient implementation with the advantage in runtime of the naive implementation. This could be done by using the space-efficient Hierholzer algorithm only for the first recursions, at which point the Euler partitions are the largest. Then, the naive implementation would be used for all subsequent Euler partitions that need to be created.
The subgraph stack structure we implemented shows very promising results. Compared to the naive implementation (which would be a typical approach to the problem of creating and managing subgraphs) the space-efficient implementation delivers better results for every measurement we conducted. It is unlikely that there is a use-case where the space-efficient implementation would be worse than the naive implementation. Like with the rank-select implementation, having a structure that manages and creates subgraphs in this way is very beneficial for other applications. As it stands, the implementation of the subgraph stack can be used as a framework for a variety of algorithms that have a need for recursively creating and managing subgraphs. Our implementation offers a wide set of operations as well; we can translate between any two subgraphs on the stack in effectively constant time and have an operation to translate between the arc-numbers and the indices in the adjacency arrays

of the graph structures (3.8). Recall that we use a specific scheme for our reference graphs (see Figure 3.14). Depending on the algorithm that uses our subgraph stack implementation, a different scheme for reference graph choosing might be more space- or time-efficient (or both). Our implementation enables the user to easily overwrite the scheme for choosing reference graphs with his own. To emulate the functionality of a recursive algorithm that would use our implementation for the subgraph stack and Hierholzer algorithm together, we implemented the simple algorithm described in Section 4.2.5, where we compared the space-efficient and naive Hierholzer algorithm, both combined with the space-efficient subgraph-stack. Our results show a significant improvement in both space efficiency and runtime when using the space-efficient subgraph-stack, regardless of whether we use the space-efficient Hierholzer or naive Hierholzer algorithm to create Euler partitions. By far the most gain in efficiency is being created by the choice of our subgraph-stack structure. Using the space-efficient Hierholzer algorithm, while using less space compared to the naive implementation, has a large runtime overhead. In reality, the gain in space efficiency provided by the space-efficient Hierholzer algorithm is overshadowed by any gains provided by the space-efficient subgraph stack.
The goal of our implementation was to provide a framework for recursive algorithms working on graph structures, with a focus of creating a foundation for solving bipartite edge-coloring problems. Recall that bipartite edge-coloring can be used to solve complex scheduling problems, such as the global scheduling of the arrival and departure of container ships at ports (Section 2.2). Such large and complex problems have a need for space-efficient algorithms and data structures, which we want to provide. The space-efficient subgraph-stack structure we implemented can be used as such a framework. Our measurements in Section 4.2.5 simulate the runtime and space-requirements of a bipartite edge-coloring algorithm, revealing the vast gains that can be achieved by using the space-efficient subgraph-stack. With the advantages created by the space-efficient subgraph-stack implementation, the choice of either the space-efficient or naive Hierholzer algorithm only provides marginal gains. Additionally, the structures used internally for both the space-efficient Hierholzer algorithm and the subgraph-stack, such as the space-efficient implementation of a rank-select structure, are used by various other algorithms and data-structures that are being implemented for the library of space-efficient algorithms [Tec18]. It can be said that our implementation now provides the basis to implement a space-efficient algorithm for graph problems such as bipartite edge-coloring, and more.

Chapter 5

Summary

The most common question that is being asked when evaluating the quality of an algorithm is: how fast does the algorithm solve the problem? In this work our focus is on a different question: how much space does the algorithm need? It is a question that often gets neglected because speed is the most important factor in many applications. Especially in fields related to *big data*, space can become an issue. Many algorithms working on large datasets require large servers to run. Increasing the amount of space on such a server is not as cheap as increasing the amount of space available on a desktop PC or laptop. A typical set of problems are graph problems. That is why we focus on a space-efficient framework that can be used for many recursive graph algorithms. All algorithms and data-structures mentioned in this thesis are integrated in the library for space-efficient algorithms [Tec18], almost all of which are implemented as part of this thesis. Besides the library for space-efficient algorithms, there are no established libraries for any programming language that provide a space-efficient set of common structures and algorithms. Even though the library for space-efficient algorithms is in its early stages, it already provides several space-efficient algorithms and data-structures in one single library. The goal of our implementation is to provide a general framework for recursive algorithms. We specifically focus on bipartite edge-coloring. Bipartite edge-coloring is a technique that can be used to solve problems related to scheduling. There are plenty real-life applications that need to solve scheduling problems; with one such example being scheduling in logistics such as the scheduling of container ships arriving at ports, which is a complex problem that can be modeled as a large graph problem. Bipartite edge-coloring makes use of Euler partitions to recursively solve such problems. That is why we implement a structure to manage subgraphs, which we call subgraph stack, and a space-efficient algorithm to create Euler partitions of graphs, which we call space-efficient Hierholzer algorithm. The implementations in [Tec18] are the first that implement all these structures, previously described theoretically in [HKL17]. Our results stemming from thorough measurements show that the space-efficient subgraph stack provides an excellent framework for many recursive graph algorithms. It shows far better runtimes and space requirements than a typical approach. Our space-efficient Hierholzer algorithm, while being 20% slower for sufficiently large inputs in creating Euler partitions than the standard algorithm for that problem, only has 20% of the space requirements, which

means that in some cases, the standard version stops from running due to insufficient space. There are various other structures that are needed internally, which are implemented as well. Many of these structures are a vital part of other data structures that are being developed for the library of space-efficient algorithms or provide valuable functionality on their own. Conclusively, we can say that the implementing work done as part of this thesis and the remaining library for space-efficient algorithms can help to save memory and to speed up several recursive graph algorithms, for which we now provide a space-efficient framework.

Bibliography

[Gab76] GABOW, Harold N.: Using euler partitions to edge color bipartite multigraphs. In: *International Journal of Computer & Information Sciences* 5 (1976), Dec, Nr. 4, 345–355. http://dx.doi.org/10.1007/BF00998632. – DOI 10.1007/BF00998632. – ISSN 1573–7640

[GNU18] GNU PROJECT: *GNU Compiler Collection version 7*. https://gcc.gnu.org/gcc-7/, 2018, accessed 10.11.18

[GRRR04] GEARY, Richard F. ; RAHMAN, Naila ; RAMAN, Rajeev ; RAMAN, Venkatesh: A Simple Optimal Representation for Balanced Parentheses. In: SAHINALP, Suleyman C. (Hrsg.) ; MUTHUKRISHNAN, S. (Hrsg.) ; DOGRUSOZ, Ugur (Hrsg.): *Combinatorial Pattern Matching*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004. – ISBN 978–3–540–27801–6, S. 159–172

[Hie73] HIERHOLZER: Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. In: *Mathematische Annalen* 6 (1873), 30-32. http://eudml.org/doc/156599

[HKL17] HAGERUP, Torben ; KAMMER, Frank ; LAUDAHN, Moritz: Space-Efficient Euler Partition and Bipartite Edge Coloring. In: FOTAKIS, Dimitris (Hrsg.) ; PAGOURTZIS, Aris (Hrsg.) ; PASCHOS, Vangelis T. (Hrsg.): *Algorithms and Complexity*. Cham : Springer International Publishing, 2017. – ISBN 978–3–319–57586–5, S. 322–333

[Kön16] KÖNIG, Dénes: Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. In: *Mathematische Annalen* 77 (1916), Dec, Nr. 4, 453–465. http://dx.doi.org/10.1007/BF01456961. – DOI 10.1007/BF01456961. – ISSN 1432–1807

[MR01] MUNRO, J. ; RAMAN, V.: Succinct Representation of Balanced Parentheses and Static Trees. In: *SIAM Journal on Computing* 31 (2001), Nr. 3, 762-776. http://dx.doi.org/10.1137/S0097539799364092. – DOI 10.1137/S0097539799364092

[NS07] NETHERCOTE, Nicholas ; SEWARD, Julian: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *SIGPLAN Not.* 42 (2007), Juni, Nr. 6, 89–100. http://dx.doi.org/10.1145/1273442.1250746. – DOI 10.1145/1273442.1250746. – ISSN 0362–1340

[rei18a] REICHELT ELEKTRONIK GmbH & Co. KG: *Microcontroller LPC 2103 FBD48 ARM7TDMI*. https://www.reichelt.de/16-32-bit-flash-microcontroller-arm7tdmi-lpc-2103-fbd48-p68399.html?&trstct=pol_0, 2018, accessed 7.11.18

[rei18b] REICHELT ELEKTRONIK GmbH & Co. KG: *Microcontroller LPC 2103 FDB64 ARM7TDMI*. https://www.reichelt.de/16-32-bit-flash-microcontroller-arm7tdmi-lpc-2129-fbd64-p68401.html?&trstct=pol_1, 2018, accessed 7.11.18

[Tec18] Technische Hochschule Mittelhessen - Fachbereich MNI: *Library for space-efficient algorithms - SEALIB*. https://github.com/thm-mni-ii/sea/, 2018. – [Online; accessed 31.10.18]

[WKT04] Weidendorfer, Josef ; Kowarschik, Markus ; Trinitis, Carsten: A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In: Bubak, Marian (Hrsg.) ; Albada, Geert D. (Hrsg.) ; Sloot, Peter M. A. (Hrsg.) ; Dongarra, Jack (Hrsg.): *Computational Science - ICCS 2004*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004. – ISBN 978–3–540–24688–6, S. 440–447