TU WIEN Informatics

# Engineering human-in-the-loop graph drawing algorithms

## A study of vertex splitting and semantic word clouds

DISSERTATION

zur Erlangung des akademischen Grades

**Doktorin der Technischen Wissenschaften**

eingereicht von

**Anaïs Villedieu, MSc**
Matrikelnummer 12028184

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg

Diese Dissertation haben begutachtet:

| | |
|---|---|
| Seokhee Hong | Philipp Kindermann |

Wien, 4. Juli 2023

Anaïs Villedieu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

# Engineering human-in-the-loop graph drawing algorithms

## A study of vertex splitting and semantic word clouds

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

**Doktorin der Technischen Wissenschaften**

by

**Anaïs Villedieu, MSc**
Registration Number 12028184

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg

The dissertation has been reviewed by:

| | |
|---|---|
| Seokhee Hong | Philipp Kindermann |

Vienna, 4th July, 2023

Anaïs Villedieu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

# Erklärung zur Verfassung der Arbeit

Anaïs Villedieu, MSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Juli 2023

Anaïs Villedieu

# Acknowledgements

I never expected to go on such a journey, but I could have expected it to be as rewarding and challenging as it was. None of this would have been possible without the invaluable help, support, and advice I received from multiple people.

The main person I would like to express my sincere gratitude to is my advisor, Martin Nöllenburg. I learned so much from you, and you inspired me to do my best. Your guidance and knowledge helped me navigate these PhD years, and your kindness and support made them enjoyable and fruitful. Whether I was feeling stuck on a project or had some idea that might be interesting, you always pushed me to carry out my projects, and I will continue this in the future.

This thesis was greatly improved by the feedback from my reviewers, Seokhee Hong and Philipp Kindermann, and I am very grateful for the time they took to read it and for their comments. I am similarly very grateful to Jules Wulms for proofreading so much of what I have written, as well as to Soeren Nickel and Markus Wallinger.

This work would also not have been possible without all the great people I have worked with all these years. Besides the aforementioned people, I want to thank Manuel, Yun, and Martin G. who helped me learn and improve so much over our time working together. I would also like to thank all of my colleagues and co-authors, as well as the members of the AC group, for all the interesting discussions, from actual research topics, to discussing language and food, with the occasional "bee talk" stop.

The pandemic lockdowns all occurred during my time at TU Wien, and it complicated things greatly for me. But the friends I made here, Jules, Soeren, and Meli in particular, our board game nights and other outings really helped me persevere.

I also need to thank the people who helped keep me together the most, and got me there in the first place, my mom and dad. You've always let me be my own person, and I am so grateful for your patience and support while I was (and still am) figuring out who that is. Our regular calls reminded me that there's always a home for me to return to, but your trust in me gave me the assurance that I should carry onward.

I am so grateful for this whole experience, so glad I decided to do a PhD, and that I decided to do it within this group and this city. I got to meet so many interesting people, travel to so many faraway places, and grow more than I ever hoped to.

# Abstract

Algorithms have become omnipresent, and advances in computer science have only increased their reach, but their applications are limited to certain types of tasks. There exist a large number of problems for which algorithmic solutions are insufficient. For example, creative tasks involve aesthetic notions that come naturally to humans but are meaningless to a machine. Thus, to undertake these problems, we consider combining a human's expert knowledge with the computing power of the machine through human-in-the-loop algorithm design. We find that graph drawing is a natural application of this paradigm, as it combines a strong focus on aesthetic optimization goals, with hard computational problems. We choose two graph drawing problems to serve as a lens into this investigation.

We first examine traditional node-link diagrams. One of the main goal when computing a classic graph layout, is to create a drawing with a low number of crossings. While drawing planar graphs is a well understood task, most graphs do not admit crossing free drawings. Thus, there has been a lot of effort invested into finding techniques to lessen the visual load induced by crossing in graph drawings. One such technique is vertex splitting. The vertex splitting graph operation replaces a vertex by copies of itself, and partitions the neighborhood of the split vertex between its copies. While vertex splitting has seen some practical applications through straightforward heuristics, its study is mostly limited to the splitting number problem in abstract graphs. In the splitting number problem, the goal is to find a planar graph, using at most $k$ splitting operations on an input graph. We find that this known NP-complete problem is also FPT when parameterized by the number of splits. We push the theoretical investigation towards more practical motivations by first studying vertex splitting in general graph drawings. We show that this problem is NP-complete, but that its subproblems are FPT when parameterized by $k$. We additionally study vertex splitting in restricted graph classes and for target properties other than planarity. We find that the problem is already NP-complete for outerplanar graphs, but bipartite graphs can be solved efficiently when vertex orderings are given in layered drawings. We also inspect these findings for more empirical applications, and to understand how to better involve a user's knowledge to use vertex splitting to improve graph drawings through a vertex splitting pipeline, which we evaluate quantitatively.

Our second axis focuses on a naturally interactive, aesthetic focused topic, word clouds.

Word clouds are a popular text visualization technique that combines creative tasks and challenging layout problems. Given an input text, one should compute a layout of the words in the plane, in which only the main words of the text are represented and scaled according to their frequency in the text. With semantic word clouds, beyond computing just a compact layout of the word rectangles, we also assume that each word is a node in an edge weighted graph. In that graph, each edge describes the semantic relatedness of its two endpoints, meaning that high weight edges describe two words that are strongly related in the input. The task then becomes for the layout we compute to reflect the weights in that graph, by placing related words in close proximity. This problem is strongly linked to contact graphs, a large category of graph problems in which graphs are represented in the plane using geometric objects. While this problem is well studied, it is computationally hard even for many restrictions on the objective function of the input graph's class. In this thesis, we first attempt to identify tractable restrictions to this problem. We study a so-called layered layout, and find that we can maximize realized contacts when the words are laid out on two layers. But our main focus is on the human-in-the-loop question. As word clouds naturally invite human interaction, we investigate layout algorithms that combine the machine's computing abilities with the user's personal aesthetic preferences, develop a tool, and evaluate it through a user study.

In this thesis, we combine theoretical and practical considerations to design efficient graph drawing algorithms. As we largely encounter NP-hard problems, we must combine restricted problem definitions, separating hard tasks into efficiently solvable subproblems, and heuristics for some cases.

# Kurzfassung

Algorithmen sind allgegenwärtig und durch Fortschritte in der Informatik hat sich ihre Reichweite nur noch erhöht. Ihre Anwendungen sind jedoch auf bestimmte Arten von Aufgaben beschränkt. Es gibt eine große Anzahl von Problemen, für die algorithmische Lösungen nicht ausreichend sind. Kreative Aufgaben zum Beispiel beinhalten ästhetische Vorstellungen, die für Menschen natürlich, aber für eine Maschine bedeutungslos sind. Deshalb betrachten wir die Kombination des Expertenwissens eines Menschen mit der Rechenleistung der Maschine durch den Entwurf von Algorithmen, bei denen der Mensch in den Prozess eingebunden wird. Wir stellen fest, dass die Darstellung von Graphen eine natürliche Anwendung dieses Paradigmas ist, da sie eine starke Ausrichtung auf ästhetische Optimierungsziele mit schwierigen Problemen verbindet. Wir untersuchen dieses Paradigma anhand zweier Graphendarstellungsprobleme.

Zunächst untersuchen wir traditionelle Netzwerk-Diagramme. Ein Hauptziel bei der Berechnung eines klassischen Graphenlayouts besteht darin, eine Zeichnung mit einer geringen Anzahl von Kreuzungen zu erstellen. Das Zeichnen von planaren Graphen ist gut verstanden, allerdings können die meisten Graphen nicht kreuzungsfrei gezeichnet werden. Daher wurde viel Aufwand betrieben, um Techniken zu finden, die die, durch Kreuzungen in Graphendarstellungen induzierte, visuelle Belastung verringern. Eine solche Technik ist das Vertex Splitting. Die Vertex-Splitting-Graphenoperation ersetzt einen Knoten durch Kopien von sich selbst und partitioniert die Nachbarschaft des gespaltenen Knotens zwischen seinen Kopien. Obwohl in einigen praktische Anwendungen bereits einfache Heuristiken für Vertex-Splitting existieren, ist die bisherige Forschung größtenteils auf das Spaltungsnummernproblem in abstrakten Graphen beschränkt. Das Ziel des Spaltungsnummernproblems ist es, ausgehend von einem gegebenen Graphen, einen planaren Graphen mit höchstens $k$ Spaltungsoperationen zu finden. Wir stellen fest, dass dieses bekannte NP-vollständige Problem auch dann FPT ist, wenn es nach der Anzahl der Teilungen parametrisiert ist. Wir fokussieren die theoretische Untersuchung auf praktisch motivierte Ziele, indem wir zuerst das Vertex Splitting in allgemeinen Graphendarstellungen untersuchen. Wir zeigen, dass dieses Problem NP-vollständig ist, aber dass seine Teilprobleme FPT sind, wenn sie nach $k$ parametrisiert sind. Wir untersuchen auch das Vertex Splitting in eingeschränkten Graphenklassen und für Zielattribute, die nicht planar sind. Wir stellen fest, dass das Problem bereits für äußerplanare Graphen NP-vollständig ist, aber bipartite Graphen effizient gelöst werden können, wenn Knotenreihenfolgen in geschichteten Zeichnungen gegeben sind. Wir betrachten diese Ergebnisse

auch für mehr empirische Anwendungen und um zu verstehen, wie man das Wissen eines Benutzers besser einbinden kann, um das Vertex Splitting zur Verbesserung von Graphendarstellungen zu nutzen, den wir quantitativ bewerten.

Unser zweiter Schwerpunkt liegt auf einem inhärent interaktiven, ästhetisch ausgerichteten Thema, den Word Clouds. Word Clouds sind eine beliebte Textvisualisierungstechnik, die kreative Aufgaben und anspruchsvolle Layoutprobleme kombiniert. Ausgehend von einem Eingabetext sollte eine Platzierung der Wörter in der Ebene berechnet werden, bei dem nur die Hauptwörter des Textes repräsentiert und entsprechend ihrer Häufigkeit im Text skaliert werden. Bei semantischen Word Clouds geht es darüber hinaus nicht nur darum, ein kompaktes Layout der Wortrechtecke zu berechnen, sondern es wird auch angenommen, dass jedes Wort ein Knoten in einem kantengewichteten Graphen ist. In diesem Graphen beschreibt jede Kante die semantische Beziehung ihrer beiden Endpunkte, was bedeutet, dass Kanten mit hohem Gewicht zwei Wörter repräsentieren, die im Eingabetext stark miteinander verbunden sind. Die Aufgabe besteht dann darin, im von uns berechneten Layout die Gewichte in diesem Graphen wider zu spiegeln, indem verwandte Wörter nah beieinander platziert werden. Dieses Problem ist eng mit Kontaktgraphen verbunden, einer großen Kategorie von Graphenproblemen, bei denen Graphen in der Ebene mit Hilfe von geometrischen Objekten dargestellt werden.

Obwohl dieses Problem gut untersucht ist, ist es selbst bei starken Einschränkungen der Graphklasse des Eingabegraphens berechnungsaufwendig. In dieser Arbeit versuchen wir zunächst, geeignete Einschränkungen für dieses Problem zu identifizieren. Wir untersuchen eine sogenannte gestufte Anordnung und stellen fest, dass wir die tatsächlichen Kontakte maximieren können, wenn die Wörter auf zwei Ebenen angeordnet sind. Aber unser Hauptaugenmerk liegt auf der Frage des menschlichen Eingreifens. Da Word Clouds natürlicherweise zur menschlichen Interaktion einladen, untersuchen wir Layoutalgorithmen, die die Rechenleistung der Maschine mit den ästhetischen Vorlieben des Benutzers kombinieren. Wir entwickeln ein Werkzeug und evaluieren es anhand einer Benutzerstudie.

In dieser Arbeit kombinieren wir theoretische und praktische Überlegungen, um effiziente Graphzeichnungsalgorithmen zu entwerfen. Da wir hauptsächlich auf NP-schwere Probleme stoßen, müssen wir eingeschränkte Problemdefinitionen, die Aufteilung von schwierigen Aufgaben in effizient lösbare Teilprobleme und Heuristiken für einige Fälle kombinieren.

# Contents

CHAPTER 1

# Introduction

Powerful computers are everywhere nowadays, even just our cell phones hold considerable computing resources. The computer's potential is unlocked through algorithms, they allow us to find routes on maps, generate summaries of text, store and retrieve information and much more. All of our software is made of a set of basic step by step instructions which allow a machine to generate a desired output from an input. Algorithms can be traced back to ancient Greece with Euclid's formula, one of the first recorded example of an algorithm. Modern advances in computer science have significantly expanded their importance, and allowed us to automate daily tasks, finding subway connections or shopping over the internet are some of the many processes now assisted by algorithms. We are therefore constantly improving their design, efficiency and ability to solve new problems, not only to make our lives more comfortable, but to also push our understanding of informatics. Algorithms allow us to solve complex problems at scale, in a time efficient manner and hence have significantly impacted the modern world.

Naturally, there exist limitations to the problems that can be solved by algorithms, and not every task can be successfully automated. There are many problems that we, as humans, solve using creativity or unconscious processes. While natural for us, they cannot be easily translated into computer instruction. There have been attempts at emulating human problem solving skills with machines, tools involving artificial intelligence (AI) and machine learning have been developed for image recognition, translation, or even to create art. Their applications are widespread, but these tools have generally been received poorly by the general public [Hic13] and currently pose ethical and legal problems [Vin23]. While automated solutions cannot not be universally applied, they nevertheless are often highly desirable. They relieve us from repetitive tasks, and allow us to focus on complex problems. Certain intricate tasks, such as design, depend on human expertise, which means they are unsuitable for large-scale automation by algorithms or AI. As an alternative, we could focus on developing solutions for specific sub-tasks by integrating expert knowledge with the computing power of machines, with the aim of simplifying tasks for designers.

Ideally, one would desire to combine the knowledge of the human expert with the capabilities of a machine, using algorithm engineering: the process of designing, evaluating and refining algorithms to better suit our need. Humans are very capable at dealing with tasks like pattern recognition, language and subtext, and are more apt at working with incomplete data or instructions, but are also less reliable and more prone to errors than machines. The notion is not novel, and the human-computer relationship is at the heart of research on human-centered computing and human-computer interaction, but neither field answers this need exactly. In human-centered computing, the human user acts as an oracle in the resolution of the problem, and human computer interaction is focused on the design of user interfaces. Our main question is how to exploit human knowledge to refine our algorithms to create optimal results that consider human preferences.

More recently, the advent of machine learning has lead to attempts to emulate human thinking through machine learning methods. These algorithms have a wide array of applications that were previously limited to human problem solving abilities, but they act as a black box. Understanding how these methods compute their outcome is a challenging task, but it is also important to note that these methods are also not correct all of the time: usually the quality of a machine learning algorithm has to be expressed through the probability it will compute the correct outcome. Additionally, these methods are highly depend on having a large quantity of high quality training data. Therefore, these methods are hard to evaluate, and, like with heuristics, we are unable to provide exact guarantees as to the quality of the solution they compute.

There exist many problems that require exact solutions, or solutions that are of sufficiently high quality (i.e. close enough to the optimal result). While other criteria, like runtime or memory usage, are important factors in judging the quality of an algorithm, we are first and foremost concerned with its correctness, or the optimality of its result. More classic algorithms are often based on formal mathematical models. We, as humans, are able, using variables, to describe as many aspect of the input as we want, and often have some quality metric which to evaluate the output. But when creating such a model, we have to abstract reality to an extent, to be able to translate into machine instructions. Modelling is necessary to describe an optimization problem, but it also creates a gap between the real world problem and the problem solved by the machine. The more variables we introduce to a model, the more difficult it becomes to understand, or evaluate it, and the less variables we use, the more our problem becomes removed from reality. While we are sometimes able to create a model that appropriately describes a problem, in other cases we have to resort to simplifying our model, and can then rely on a human-in-the-loop approach to let a human assist where the model deviates too much from its reality.

In this thesis, we apply this algorithmic methodology to the field of graph drawing, which deals with the study of visualization of complex and large network data sets using the simple structure of a graph. As data sets become increasingly large and complex, there is a growing need to comprehend, analyze, and communicate their information effectively, which motivates our investigation into good methods for graph drawing. Graph drawing is especially interesting for this human-in-the-loop application. Graph drawing not only

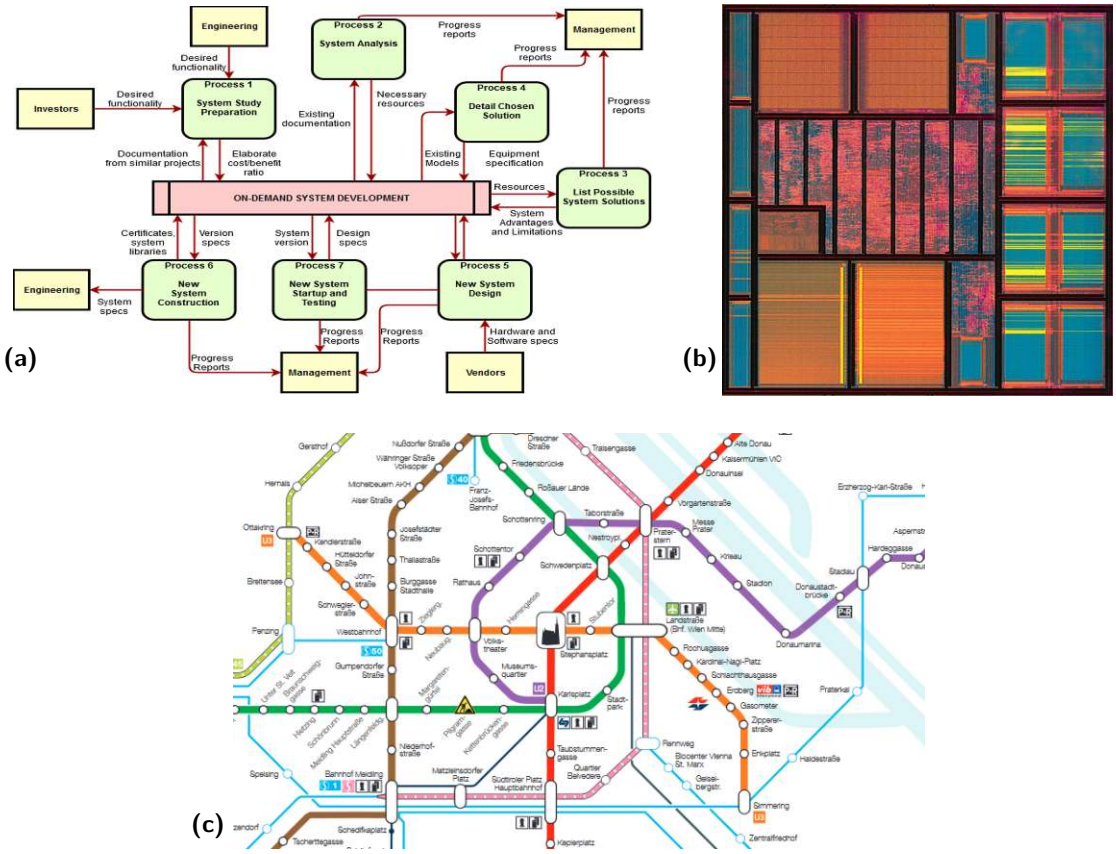



Figure 1.1: Examples of graph drawings, **(a)** a software diagram where nodes are drawn as boxes and edges are directed and rectilinear, **(b)** a VLSI integrated circuit where each rectangular component can be modeled as a node in a graph and adjacency between two components can be modeled by an edge, and **(c)**, an example of a schematic map, the Viennese metro system, edges can only be vertical, horizontal and diagonal to represent the real world topology of the subway network.

involves constructing an accurate geometric representation of a network, but to also create and aesthetically pleasing and intelligible image. These aesthetic and intelligibility aspects are something that any human can intuit, but machines have no direct concept of, unless we manage to model them appropriately. Graph drawing has a wide range of applications, ranging from network analysis and software and database diagrams (see Fig. 1.1(a)), to very large-scale integration (VLSI) for chip designs (see Fig. 1.1(b)), schematic mappings (see Fig. 1.1(c)) and more.

Usually graph drawings have to conform to a set of drawing conventions (hard constraints), for example technical diagrams more often use rectilinear edges, and cartograms use particular shapes instead of points. To efficiently compare drawings that adhere to the desired aesthetic conventions, a large set of optimization problems use additional soft

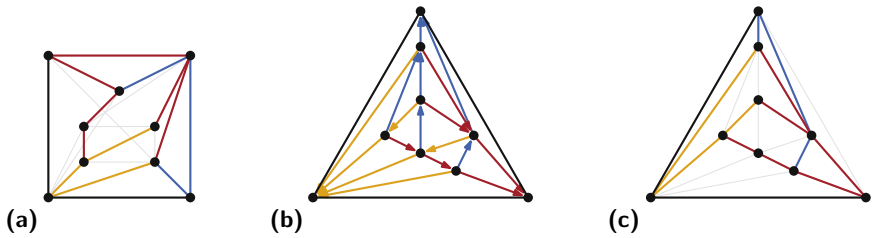



Figure 1.2: **(a)** A graph drawn manually without crossings, **(b-c)** the same graph drawn using a Schnyder realizer [Sch90], the automated method achieves a crossing free drawing but has sharp angles and lacks the symmetry of the hand drawn version. The edges drawn in light gray represent the edges added to compute the Schnyder realizer.

constraints. For example, given a graph that can be drawn without crossing, we would look for such a drawing, while trying to minimize the overall area of the drawing. Other important metrics that try to capture graph aesthetics are the angular resolution, the edge-length ratio, the number of bends, or the symmetry of the drawing, and many more.

While algorithms that achieve these aesthetic goals have been designed, automated methods cannot target every existing quality metric that has been defined at the same time? Commonly a relevant subset of these criteria is selected for optimization, which can often lead to sub-par layouts, as shown in Fig. 1.2. Additionally, automated methods are largely non-interactive, meaning they can produce a layout but will not allow for fine tuning, or for small changes to the objective function. In these areas, human interference should be considered, as humans are largely able to incorporate minute changes to optimize the aesthetic appeal of the graph. However, creating a good layout from scratch for a large data set is a very challenging task for humans (see Fig. 1.3). Instead, humans are more suited to edit an existing layout, for example by identifying which parts of an existing drawing are more readable, better looking, and which ones need to be improved. Thus, we often require automated methods like force-based algorithms Human perception can handle many aesthetic criteria in parallel, but can usually not optimally target any specifically. The abilities of the human, and those of the algorithms can theoretically synergize to create graph drawings of high quality.

This makes graph drawing an ideal candidate field to study the question of human-in-the-loop algorithmic design. This connection is fairly natural, and there has been a large interest especially for more general graph drawing challenges like directed layered graphs [dNE01], network diagrams or orthogonal layouts [DMW08] where the user interacts with the drawing to produce visualizations of high quality. This problem has also been used in more specialized settings [KDMW16], where the knowledge of human designers was used to refine anatomical drawing labeling algorithms. As most general graph drawing problems are highly complex and often can consider a very large number of quality criteria, we choose here to study more specific problems.

Graph drawing offers a broad diversity of questions which each can be differently understood through the lens of human-in-the-loop algorithm engineering. In this thesis, we will

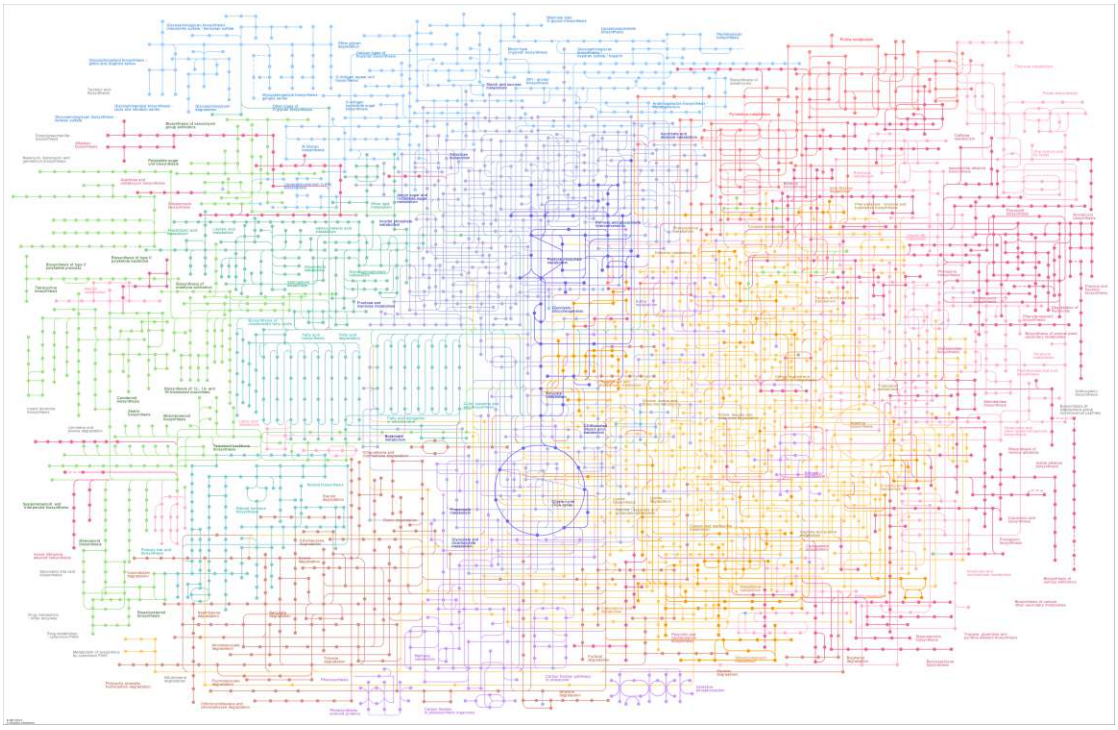


Figure 1.3: The KEGG metabolic pathway is a large manually created graph drawing. Biological pathways are usually dense graphs and are challenging for automated layout methods, therefore significant expert effort is required to create these layouts.

focus on two types of problems to better understand this paradigm. We will first look at traditional node link graph visualization and crossing minimization, through the study of vertex splitting, and later consider the highly interactive and playful problem of word cloud layouts. We study these problems initially from a theoretical perspective through the study of algorithms and complexity to familiarize ourselves with the challenges each class of problem offers, and to understand what solutions currently exist. With this knowledge, we then initiate the human-in-the-loop considerations. In this second step, we study where our automated methods might be improved by the knowledge of a human user, and develop our algorithms further to be able to consider this additional human input. We will first properly introduce the two main topics of these thesis, namely vertex spitting and semantic word clouds.

## 1.1 Background

In this section we define and motivate both the vertex splitting and semantic word cloud problems more closely. We cover the related literature, focusing on both state-of-the-art theoretical and algorithmic results, as well as relevant practical approaches.

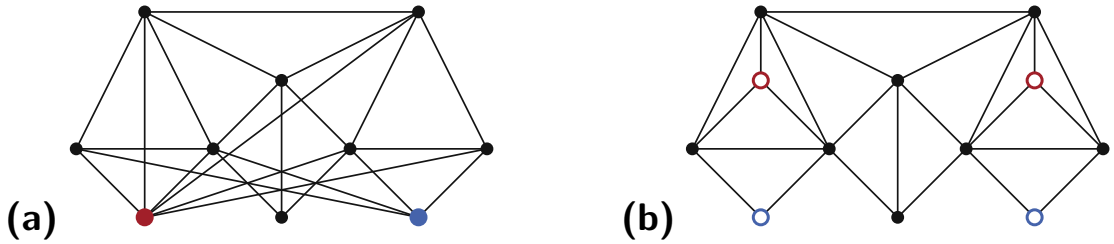



Figure 1.4: **(a)** Drawing with ten crossings, and four instances of three edges crossing on the same point due to symmetry, **(b)** the planar drawing obtained after two splitting operations.

### 1.1.1 Vertex Splitting

One of the main challenges of network visualization, is to compute high quality layouts of dense data sets. Large graphs, or "hairball"-like graphs, are often difficult to display, as they contain a lot of visual clutter like edge crossings [vLKS+11]. Plane (or almost plane) drawings on the other hand are simpler to read and preferable for any audience: Edge crossings are known to be highly detrimental to the readability of a graph [Pur00, PCA02, PPP12]. Therefore, it is important to find a drawing of a graph with a low number of crossings. If the graph is planar then finding a plane drawing is simple, but in the general case, computing a graph drawing with a minimal number of crossings is NP-complete [GJ83].

This issue has motivated a lot of research into efficient ways of dealing with crossings in non-planar graph drawing [Nö20, Lie01, Sch18]: some methods attempt to find drawings which achieve a small number of crossings [PT97, DLM19], while others focus on limiting the negative visual impact of the crossing on the readability of the drawing. This can be achieved by, for example, bundling edges [LHT17, FHSV16, ZXYQ13]. With edge bundling, some edge segments running close to one another in a similar direction are drawn together in a bundle, and when those bundled edges cross an edge, multiple crossings then appear as a singular crossing. Alternatively, one can also focus on improving the crossing angle [Oka20], usually striving for right angle crossings [Sch21], which make the pair of crossing edges easier to distinguish and make paths in the graph easier to follow [DEL11, AFK+12]. Other methods attempt to obtain planar drawings from non planar ones, by for example removing edges or vertices, as with vertex deletion [MS12, LY80] or the problem of finding a maximal planar subgraph [JM96]. Analogously, when considering graph thickness or simultaneous embeddings, we attempt to partition a non planar drawing into a set of plane graphs that share vertices. We chose in this thesis to investigate a method known as vertex splitting to facilitate the visualization of non planar graph drawings: Considering an input graph $G = (V, E)$, a vertex split of vertex $v \in V$ is done by removing $v$ from $V$ and inserting instead two new vertices $v_1, v_2$ to $V$, the *copies* of $v$. The previous neighborhood $N(v)$ of $v$ is then partitioned between $v_1$ and $v_2$.

As an operation, vertex splitting can be used to achieve many graph properties: consider a graph that lacks a specific property $\Pi$ (e.g. cycle-free, planar, admits a tension-free layout) we can apply a vertex splitting procedure repeatedly until we obtain a graph with property $\Pi$. Splitting vertices has also been studied in practical setting. For example, when visualizing large graphs, in biology with metabolism networks [NOM+19, WNSV19, WNV20], social networks [HBF08], or set membership [HRD10].

In our case, we focus on the resolution of edge crossings Fig. 1.4. This application is not novel, it is strongly linked with the *splitting number* graph invariant. The splitting number of a graph $G$ is the minimum number of vertex splitting operations that are required on $G$ to obtain a planar graph. Splitting number was introduced as a method to measure how close a graph was to a planar graph. When dealing with graph drawings, the idea of vertex splitting for crossing minimization is intuitively sound. Crossings are not desirable, therefore splitting vertices to remove those crossings allows us to achieve a more readable drawing that still retains all of the original embedding's information. On the other hand, every split adds an additional vertex to the drawing, this increases the number of objects to keep track off and creates clutter. Additionally, the connectivity of the graph becomes partially hidden as some paths become harder to follow. Therefore, our main task in this thesis was to understand how vertex splitting extends to graph drawings. One of the main questions we asked ourselves was, when given a drawing, how many splits are required to obtain a plane drawing, and how to obtain this plane drawing. Computing the splitting number is known to be NP-complete for general graphs (even with maximum degree 3) [FdFdMN01], and bipartite graphs [AKK22]. For some restricted graph classes the splitting number is known, namely for the complete graph [HJR85], the complete bipartite graph [JR84] and the 4-Cube [FdFdMN98]. The splitting number has also been studied beyond the plane, on different surfaces [Har86, Har87]. Intuitively, one could guess that the additional information given by the drawing in the embedded setting could give sufficient hints to find the necessary splits to obtain a plane drawing, but we find that the task also presents an additional set of challenges. Vertex splitting on a drawing can be realized through the five following steps (see Fig. 1.5):

i Choosing a set of vertices to be split,

ii Choosing how many times each vertex should be split,

iii Choosing a partitioning of the original vertex's neighborhood,

iv Choosing an embedding for each copy in the remaining drawing,

v Routing the edges between the copies and their neighborhood partition.

Note that step (iii) and (iv) are interchangeable, and in the case where we aim to achieve planarity rather than crossing minimization, step (v) can be trivial. Additionally, once the neighborhood of a copy is known, if planarity is the goal then the embedding question for that copy should to be solved easily. However, for multiple copies one must carefully

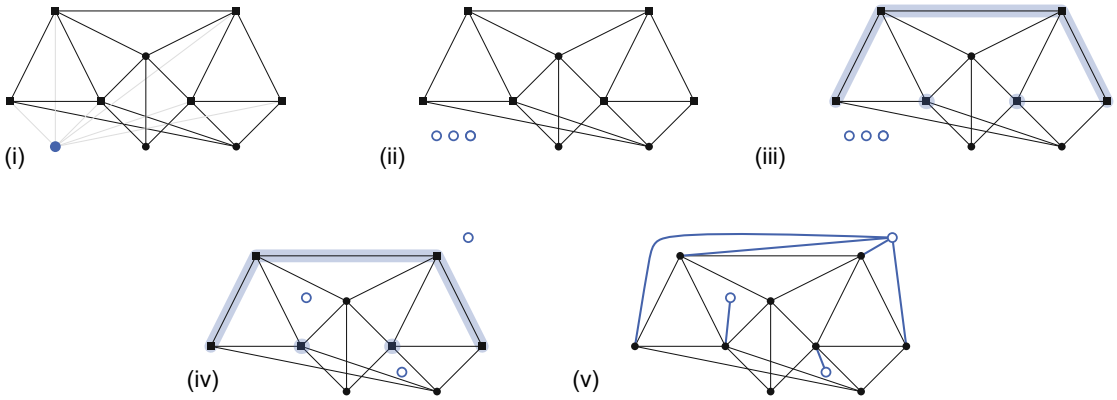



Figure 1.5: The five steps of the vertex splitting pipeline, the vertex drawn as a blue disk in the first drawing is split twice into the three copies drawn as blue circles. One copy is placed in the outer face and has four neighbors, the two other copies have degree one and are connected to inner vertices.

consider how they interact with one another and prevent new crossings from being induces by edges incident to copies.

We briefly consider a different optimization problem, where rather than minimizing the number of splitting operations, one instead focuses on minimizing the number of input vertices that will be split. In essence, in this setting, each vertex selected to be split can be split $\delta(v)$ many times, where $\delta(v)$ is the degree of the split vertex $v$, with no additional detriment to the optimization function. This operation is sometimes referred to as a *vertex explosion*. We find that this setting closely resembles the vertex deletion problem, where we want to decide whether a given graph can be made planar by deleting at most $k$ vertices, and to related problems of hitting graph minors by vertex deletions. Given the smallest set of vertices to be removed from a graph $G$ to obtain a planar subgraph $G'$, we only need to reinsert $\delta(v)$ many copies of each removed vertex to the graph, which does not disturb planarity as those vertices have degree one. Additionally, we note here that the subproblem of reinserting vertices into planar graphs also shares similarities with drawing extension problems, where a subgraph is drawn and the missing vertices and edges must be inserted in a (near-)planar way into this drawing [ADBF$^+$15, CGMW09, CH16, EGH$^+$20a, EGH$^+$20b]. In extension problems, the set of vertices to be inserted, as well as their precise neighborhood, is usually known, unlike in the vertex splitting problem setting. To our knowledge, vertex deletion has not been studied when the input is a drawing, therefore the complexity question of the vertex minimization problem formulation is not trivial, but we note here that many graph extension problems are NP-hard when crossing minimization is a consideration [AKP$^+$20].

There exist similar problems to splitting number that rely on vertex splits: one close example is the split thickness of graphs [EKK$^+$18], also called the folded covering number [KU16]. In this setting, any number of vertices can be split, and they can be split at most $k$ times. One must then find the smallest $k$ that will result in a planar graph.

### 1.1.2 Semantic Word Clouds

Word clouds are a classical information visualization technique through which textual data is displayed in an aesthetically pleasing and engaging manner. In a word cloud visualization, a set of words extracted from an input text are displayed in a compact fashion, and each word's font size is scaled according to its frequency in the input text. It is a standard method to abstract, visualize and compare textual data [VWF09]. It was initially known under the name tag clouds [VW08], and was often used to summarize web pages. Word clouds were also used to analyse and compare speeches of political candidates [Sch08], shortly before gaining traction with the general public with the tool Wordle [Fei09, VW08]. Wordle is an automated word cloud layout web hosted system, that heavily popularized word clouds. It also generated a lot of research interest, initially within the visualization community [KLKS10, WCB+18, CWL+10], which later extended towards graph drawing and computational geometry [BFK+14, BCL+16].

Most initial algorithms to compute word cloud layouts focused on creating tightly packed visualizations, using forced based systems or spiral placement [VWF09]. New tools were later introduced that were aimed at integrating interactions. Users could to color, move and rotate words to create more desirable visualizations [KLKS10, WCB+18]. While their popularity was notable, word clouds were also criticised as a information visualization technique, and found to not be effective at delivering the underlying information of the input text [HPP+20]. While a word cloud informs its audience about term frequency, it does not reflect the connections of the different words. Hearst et al. [HPP+20] found that while regular word cloud did not display and communicate the text data well, semantically grouped layouts were significantly more effective, meaning one should strive to represent related words by embeddings these words closely together in the plane.

The criticisms of the classic compact word cloud layout motivated a new class of layout algorithms, namely, *semantic word cloud layouts*. In such a layout, the relative positioning of the words to one another is meant to carry meaning. The first algorithm that encoded these relationships within the display was proposed by Cui et al. [CWL+10]. They proposed multiple methods to evaluate semantic relatedness and used them to lay out the words in the plane. With their methods, similarity vectors were created for each word, that described the word's relationships with all other words. Using multi-dimensional scaling (MDS) and this set of vectors, the words were laid out in the plane. As compactness remained a concern to obtain aesthetically pleasing layouts, a force system was used to finalize the drawing by bringing the words closer together in a tight packing.

More theoretical approaches were also considered: Barth et al. [BFK+14] introduced the Compact Representation of Word Network (CROWN) problem to create algorithms with quality guarantees for these layout problems. They described a new optimization metric to capture semantic relatedness by requiring that two strongly correlated words touch in the representation. The CROWN problem takes as input an edge weighted graph, where the edge weight between two nodes corresponds to how strongly related the two corresponding words are. For every edge that is realized as a proper contact between two


Figure 1.6: The graph and its representation as a contact graph, the word "Switzerland" can only realize three out of its four adjacencies due to the fixed aspect ratio.

axis aligned rectangular boxes, a profit equivalent to the weight of the edge is gained. The task is to find a layout of the rectangular word boxes that maximises the total profit of the layout, as shown in Fig. 1.6. They found that this problem was NP-hard, but proposed many approximations for restricted classes of graphs.

Word cloud layouts are similar to rectangular contact graphs, where a graph must be represented as a set of rectangles that touch if they share an edge. Similarly, in area universal rectangular layouts, a rectangle is partitioned into a set of interior-disjoint rectangles. The dual of this representation is called a rectangular dual, and such a graph is necessarily planar and has no separating triangle [KK85]. The areas of the rectangles to be realized in the representation are also part of the input. These layouts find application in cartograms, floor plans or VLSI design. However, in the case of semantic word clouds, not only does the area of the boxes have to be respected, but their aspect ratio is set as well. Additionally, the graphs representing semantic relatedness can be obtained using natural language processing (NLP) from input texts are often very dense, heavily non planar, thus realizing all contacts is usually impossible.

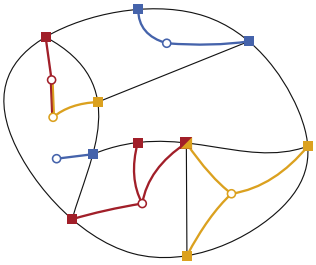
While semantic word clouds have been well studied, they have surprisingly not been considered in an interactive or human-in-the-loop setting, unlike traditional word cloud layouts. Barth et al. outlined several different optimization tasks for the CROWN problem, and found that they often were hard to optimize [BFK+14], therefore, finding efficient-human-in-the-loop algorithms for the semantic word cloud problem is a challenging task that should provide good insight into our paradigm.

## 1.2 Thesis Outline

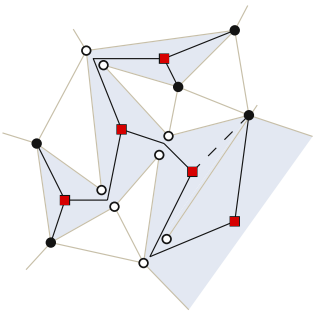
The main body of the thesis is centered around the two themes introduced above, which each require a different approach to our human-in-the-loop consideration. As previously stated, in the first part we will study the vertex splitting graph operation and in the second part, we will look at semantic word clouds. In both sections, our discussion is initiated with a theoretical investigation. We first present algorithmic and complexity results, then explore how to extend this knowledge to human-in-the-loop algorithmic considerations.

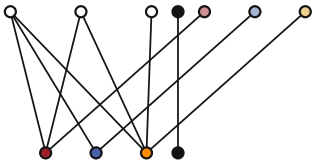
**Part I - Vertex Splitting** Vertex splitting has been studied mainly through the lens of the splitting number. While results are known for the splitting number of specific graph classes, there exists currently no algorithm to compute it or approximate it. There exist limited algorithmic results that use vertex splitting, these have mostly been used heuristically in the context of tension layouts or in the context of biological network visualization.

**Chapter 3 - General Graph Drawings** The splitting number of a graph $G = (V, E)$ is the minimum number of vertex splits required to turn $G$ into a planar graph. It is known to be NP-complete for abstract graphs. In this chapter we provide a non-uniform fixed-parameter tractable (FPT) algorithm for this problem. We then shift focus to the splitting number of a given topological graph drawing in $\mathbb{R}^2$, where the new vertices resulting from vertex splits must be re-embedded into the existing drawing of the remaining graph. We show NP-completeness of this *embedded* splitting number problem, even for its two subproblems of (1) selecting a minimum subset of vertices to split and (2) for re-embedding a minimum number of copies of a given set of vertices. For the latter problem we present an FPT algorithm parameterized by the number of vertex splits. This algorithm reduces to a bounded outerplanarity case and uses an intricate dynamic program on a sphere-cut decomposition.

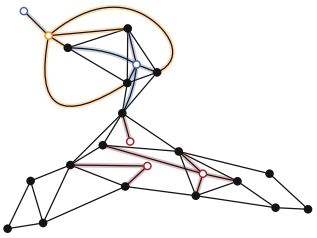


**Chapter 4 - Planar Drawings** As many problems in graph drawings are hard, we choose to restrict them to a limited input, often by considering special graph classes. A natural question that arises from Chapter 3 is to wonder if there exists a class of graphs that can be split to achieve a certain property in polynomial time. Here we study how to minimize the number of splits to turn a plane graph in graph class $\mathcal{G}$ into an outerplane one. We tackle this problem by establishing a direct connection between splitting a plane graph to outerplanarity, finding a connected face cover, and finding a feedback vertex set in the graph's dual. We prove NP-completeness when $\mathcal{G}$ consists of plane biconnected graphs, while we show that a polynomial-time algorithm exists for the class of maximal planar graphs. Additionally, we provide upper and lower bounds on the minimum number of splits required for certain families of maximal planar graphs. Lastly, we propose a SAT formulation that computes an optimal set of splits to be applied to an instance to transform any abstract graph into an outerplanar graph, and gives an ordering of vertices for its outer face.





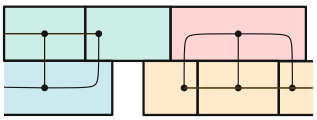
**Chapter 5 - Bipartite Graphs** Bipartite graphs model the relationships between two disjoint sets of entities and are naturally drawn as 2-layer graph drawings. In such drawings, the two sets of entities (vertices) are placed on two parallel lines (layers), and their relationships (edges) are represented by segments connecting vertices. Methods for constructing 2-layer drawings often try to minimize the number of edge crossings, and this minimization question is also the focus of the chapter. Indeed, while planar drawings have often very useful properties, vertex splits introduce significant noise to a drawing. Thus, we focus on the following question: given a fixed budget of splits, can we minimize the number of crossings of a bipartite graph where the vertex ordering of one of the bipartitions is given. We find again that this problem is NP-complete, but propose an XP-time algorithm to solve the problem.



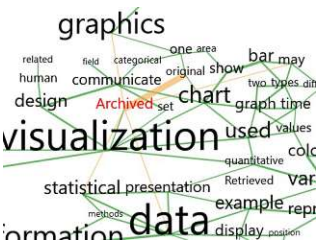
**Chapter 6 - Practical Consideration** When visualizing complex graphs, a large consideration is placed on dealing with crossings. While a lot of attention has been given to aggregation techniques to compute legible visualizations, these techniques largely do not allow for a close reading of the underlying data they represent. Vertex splitting, on the contrary, allows for such a view, while similarly being a powerful tool for crossing reduction in drawings. Although it has a strong theoretical foundation, it has, to our knowledge, not yet been closely studied for practical applications. In this chapter, our goal is to begin this discussion, by introducing novel metrics to evaluate vertex splitting algorithms, as well as defining design guidelines and outlining algorithmic considerations for this problem. As vertex splitting is a complex operation, we focus on three main subproblems, selecting which vertices should be split, computing the neighborhood of the split vertices' copies and embedding the copies into a given drawing. We present a set of efficient algorithms for each step of this pipeline, that encapsulate different drawing considerations, and are compatible with constraints set by the user.

**Part II - Semantic Word Clouds** Word clouds naturally invite the human-in-the-loop considerations. There have been many interactive tools developed that let a user design their own word clouds. Classical word clouds though, have limited interest when it comes to designing algorithms with guarantees as they are limited to considerations of compactness in terms of optimization. We instead focus on semantic word clouds, which provide us with many interesting questions to solve.

**Chapter 7 - Layered Semantic Word Clouds** First, we consider a restriction on the input graph to the CROWN problem (Compact Representation of Word Network) by considering layered, embedded planar graphs. We investigate two optimization problems on area-proportional rectangle contact representations for these graphs. We represent the vertices as interior-disjoint unit-height rectangles of prescribed widths, grouped in one row per layer, and each edge is ideally realized as a rectangle contact of positive length. In this chapter, we want to maximize the number of realized rectangle contacts or minimize the overall area of the rectangle contact representation, while avoiding any false adjacencies. We present a network flow model for area minimization, a linear-time algorithm for contact maximization of two-layer graphs, and an ILP model for maximizing contacts of k-layer graphs.



**Chapter 8 - Interactive Semantic Word Clouds** In this chapter we present MySemCloud, a new human-in-the-loop tool to visualize and edit semantic word clouds. MySemCloud lets users perform computer-assisted local moves of words, which improve or at least retain the semantic quality. To achieve this, we construct a word similarity graph on which a system of forces is applied to generate a compact initial layout with good semantic quality. The force system also allows us to maintain the compactness and semantic quality of the layout after each user interaction, as well as preserve the user's mental map. The tool provides algorithmic support for the editing operations to help the user enhance the semantic quality of the visualization, while adjusting it to their personal preference. We show that MySemCloud provides high user satisfaction as well as permits users to create layouts of higher quality than state-of-the-art semantic word cloud generation tools.



CHAPTER 2

# Preliminaries

In this chapter, we will introduce the fundamental notions that will be used throughout this thesis. We start by defining some core notions of graph theory, and graph drawing in Section 2.1. With these definitions, we introduce in Section 2.2, the main classes of graphs that will be used in later chapters. Lastly in Section 2.3 we will cover some general notions of algorithmic complexity, as well as algorithm design.

This chapter is not an exhaustive introduction of each of the covered themes, but rather focused on the topics that are relevant to this thesis. We refer to more comprehensive literature in each of the following sections.

## 2.1 Graphs and their Drawings

A *graph* $G = (V, E)$ consists of a set of *vertices* $V$ and *edges* $E$. An edge $e \in E$ corresponds to a pair of vertices $u, v \in V$. We write $V(G)$ for the set of vertices of graph $G$, and we similarly write $E(G)$ for $G$'s edges. For $e \in E$, $e = (u, v)$ we call $u$ and $v$ the *endpoints* of $e$, and we say that $e$ is *incident* to its endpoints. Additionally, we say that two edges are *adjacent* if they share an endpoint, or two vertices $u, v$ are adjacent if $(u, v) \in E$. In the case of a *directed* graph, an edge $(u, v)$ is an ordered pair, and $u$ is adjacent to $v$, but the reverse is only true if and only if the edge $(v, u)$ exists. In the general case, when an edge $(u, v)$ is an unordered pair, we call the graph *undirected*. For a vertex $v \in V$ we call the *degree* of $v$ the number of edges that are incident to $v$. We refer to the *neighborhood* of $v$ as the set of vertices that $v$ is adjacent to, and write $N_G(v)$ for the neighborhood of $v$ in $G$, or drop the subscript when $G$ is clear from context. Consider $H = (V_H, E_H)$, we call $H$ a *subgraph* of $G$ if $V_H \subseteq V$ and $E_H \subseteq E$ and $(u, v) \in E_H$ if and only if $u, v \in V_H$. We call the graph $G' = (V', E')$ an *induced subgraph*, where $V' \subseteq V$, and $E' = \{(u, v) \mid u, v \in V'\}$, and we write $G' = G[V']$.

In an undirected graph, if $e = (u, v)$, $e' = (u, v)$ and $e \neq e'$ then $(u, v)$ is a multi-edge. If $e = (u, v)$ and $u = v$, then $e$ is a *loop*. These definitions are necessary to introduce the fundamental notion of *simple graphs*. A graph is a simple graph if it is an undirected graph with no loop or multi-edge. In this thesis, unless specifically stated otherwise, every graph is a simple graph. The last important combinatorial notion is the notion of *connectedness*. We say that a graph is *connected* if for any vertex pair $v_1, v_k \in V$, there exists an ordered set of vertices $P = \{v_1, \ldots, v_k\}$ such that for any $1 < i < k - 1$, $(v_i, v_{i+1}) \in E$. Additionally, we say that $P$ is a *path* of length $k$. If such a path does not exist for every vertex pair we say that the graph is disconnected, and each connected subgraph is a *connected component*.

A drawing $\mathcal{D}$ of a simple graph $G = (V, E)$ is a mapping of the vertices of $V$ to points in the plane and of the edges of $E$ to curves such that for an edge $(v, u) \in V$ the endpoints of the corresponding curve are the points that correspond to the vertices $u$ and $v$. In this thesis, the notions that have been introduced for graphs are extended as is for drawings, meaning we regularly refer to the vertices of a drawing, or their adjacency and so on. In a *simple drawing*, two curves may only intersect once, either on their endpoints or in their interior, meaning that adjacent edges do not cross on their interior, and any edge can cross a non adjacent edge only once. We say that a drawing is *plane* if no two edges intersect each-other on their interior. In a plane drawing, we define the *faces* of the drawing as regions of the plane that are bounded by a set of edges. We say that edges and vertices are incident to a face that they bound, or that two faces are adjacent if they share an edge or a vertex on their boundary. When given a drawing $\Gamma$ of a graph $G$, we can compute a dual graph of $G$, the dual graph $D = (V_D, E_D)$ is obtained by placing a vertex in every face of $G$ in the drawing $\Gamma$, and for each edge in $\Gamma$, adding an edge in $D$ between the vertices (or vertex) that correspond to the faces that are incident to that edge in $\Gamma$.

A graph can also be represented in the plane with geometrical objects. We call a drawing of $G$ a *contact graph* if the vertices of $G$ are represented by objects (segments, rectangles, circles), and the edges of $G$ are represented by two objects touching, or overlapping.

A more complete introduction to graph theory can be found in the book by Diestel [Die17], and graph drawing is covered more thoroughly in [BETT99].

## 2.2 Graph Classes

A graph class represents a family of graphs that all obey a certain property $\mathcal{P}$. Throughout this thesis, several graph classes will be relevant and we will introduce here the main graph classes. We previously introduced paths, a special case of a path is a *cycle*: A path $P = \{v_1, \ldots, v_k\}$ is a cycle if $v_1 = v_k$, and the path in this case forms a close loop. Note that the edges on the boundary of a face form a cycle. Another important class are *trees*, which are graphs that have no cycles. In a tree each vertex pair is connected by exactly one path if the graph is connected. Otherwise, the graph is called a *forest* and its vertices are connected by at most one path. In a tree, we commonly denote a

vertex as a *root*. Every non-root vertex of degree one in that tree is called a *leaf*. We denote the *depth* of a vertex in the tree as the length of the path between the root and that vertex. If two vertices $u$ and $v$ are connected by an edge, and $v$ is at depth $d$, then $u$ is necessarily be at depth $d-1$ or $d+1$. If $u$ is at depth $d+1$, the $u$ is a *child* of $v$, and if $u$ is at depth $d-1$, $u$ is the *parent* of $v$.

We say that a graph $G = (V, E)$ is a *complete* graph if for any $u, v \in V$, $(u, v) \in E$. If $|V| = n$, then the complete graph on $n$ vertices is written $K_n$. We say that $G$ is *bipartite* if there exists two *bipartitions* of $V$: $V_1, V_2$ such that $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$, and for $(u, v) \in E$, if $u \in V_1$ then $v \in V_2$ or similarly if $u \in V_2$ then $v \in V_1$. We can combine the two previous notion to obtain *complete bipartite graphs*. If $|V_1| = n$ and $|V_2| = m$, then we write $K_{n,m}$.

A *planar* graph is a graph that has an embedding in the plane where no curve intersects another curve on their interior, meaning it admits a plane drawing. The previously introduced notions of complete graphs can also be used to characterize plane graphs using Kuratowski's theorem, which dictates that a graph is planar if and only if it does not contain a subgraph that is a subdivision of the $K_5$ or the $K_{3,3}$. *Outerplanar* graphs are a specific class of planar graphs, that admit a drawing where there exists a face to which every vertex is incident to. Commonly this face is the unbounded face of the drawing, called the *outer face*. This notion is extended to $k$-outerplanar graphs, a $k$-outerplanar graph admits a drawing where every vertex on the outer face can be removed to obtain a $(k-1)$-outerplanar graph, and an outerplanar graph is a 1-outerplanar graph.

## 2.3 Complexity and Algorithms

This thesis mainly focuses on algorithms, thus, complexity results are mostly limited to the complexity classes P and NP, which are a small subset of the classes investigated in the broad field of computational complexity. For a comprehensive introduction to complexity theory, interested readers can refer to "Introduction to the Theory of Computation" by Sipser [Sip97] or the more recent book by Arora and Barak [AB09]. For the topic of parameterized complexity, the book "Parametrized Algorithms" [CFK+15a] provides a thorough view into the topic.

The P and NPcomplexity classes can be understood in the following manner. Consider $\Sigma$, and fixed finite *alphabet*. We call $\Sigma^*$ the set of finite sequences of symbols in $\Sigma$. A *decision problem* corresponds to a pair $x, L$, where $L \subseteq \Sigma^*$ is a *language*. The task is to *decide* if $x \in L$, in which case a deterministic Turing machine outputs the value 1 and 0 if $x \notin L$. If $x \in L$ then it is a *yes-instance* of $L$ and it is called a *no-instance* otherwise. Most importantly, a decision problem is said to be in P if there is a deterministic Turing machine that solves $x \in L$ in polynomial time. We say then that the decision problem is *polynomial time solvable*, and similarly, a problem is said to be in NP if there is a nondeterministic Turing machine that decides the problem in polynomial time. It is known that $\mathsf{P} \subseteq \mathsf{NP}$, however, the question whether $\mathsf{P} = \mathsf{NP}$ remains open and is considered one of the most important question of complexity theory and general

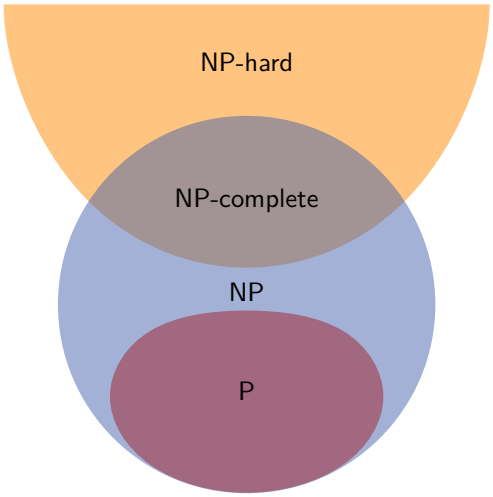



Figure 2.1: The Euler diagram representing the relationships between P, NP, NP-hard and NP-complete, assuming P $\neq$ NP.

informatics. We are most often interested in knowing if the problem we introduce lies in P or not. To show this, we make use of reductions. Consider two decision problems $x, L$ and $x', L'$. We say that $x, L$ is polynomial time reducible to $x', L'$ if there is a polynomial time computable function that takes as input an instance $I'$ of $x, L$ and returns and instance $I$ of $x', L'$, such that $I$ is a yes-instance if and only if $I'$ is a yes-instance. We say that $L$ is *polynomial time (Karp) reducible* to $L'$. We say that a decision problem is NP-*hard* if every other decision problem in NP is polynomial time reducible to it. Lastly, we say that a problem is NP-*complete* if it is both in NP-hard and NP. In this thesis, as is commonly done, we show membership to the NP complexity class using a *verifier* $M$. A verifier is a deterministic Turing machine such that, for $x \in \Sigma^*$ and a polynomial $p$ such that $u \in \Sigma^{|p(x)|}$, we find that $M(x, u) = 1$ if and only if $x \in L$. Thus we show that given a solution to a decision problem, if we can decide in polynomial time that the solution is correct, then the decision problem is in NP. In this thesis, we make the common assumption that P $\neq$ NP, meaning that when membership to NP-hard or NP-complete is shown for our problem, then is it highly unlikely that an efficient, polynomial time, algorithm exists, see Fig. 2.1.

To deal with NP-complete problems, we can employ several methods to obtain exact solution. In this thesis we employ methods from mathematical optimization like *integer linear programming (ILP)*. Specifically, we model an *optimization problem* as a set of equations using the following format:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b, \\ & x \geq 0 \\ \text{and} & x \in \mathbb{Z} \end{array}$$

Here $x$ is the output, the values that should be decided, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{n \times m}$ are the input. If no value is a binary or an integer, the model can be solved in polynomial time, and the problem that is modelled lies in P, but integer programming is NP-complete. Note that if a problem can be formulated using an ILP it is not necessarily NP-complete or NP-hard. There exists *solvers* that can solve these models fairly quickly which can make them viable solutions to small instances of hard problems.

Another option to model hard problems is to make use of the *Boolean satisfiabilty problem* (SAT). In this problem, we are given a *Boolean* formula that is build from variables, parenthesis and the operators AND, OR, NOT. If one can assign the values TRUE or FALSE to each variables such that the overall formula can be made true, then this instance of SAT is satisfiable. Formulas are usually written using the *conjunctive normal form* (CNF), meaning they are written as a conjunction of clauses. A *clause* is a disjunction of literals and a *literal* is either a variable or a negation of a variable. There are many variations of SAT that are NP-hard, for example 3-SAT and MAX-2-SAT. In the first case the CNF formula is limited to at most three literals per clause, and at most two for the second, with the added rule of having to determine the maximum number of clauses that can be satisfied simultaneously by an assignment. Both settings can be used to model decision problems and can be solved with the use of solvers.

As solvers act as somewhat of a black box to solve these hard problems, we also propose another type of exact algorithmic results for NP-hard problems using *parameterized complexity*. Not all NP-hard problems are equally as difficult, and parameterized complexity can grasp some of these nuances of difficulty. In essence, we are given as input a problem and a *parameter*, that should ideally be somewhat small, and we study the complexity of the problem as a function of this parameter.

Formally, recall that $L \in \Sigma^*$ is a language over the alphabet $\Sigma$. A parameterized problem $L$ is *fixed-parameter tractable*, or in the complexity class FPT if for the parameter $k$, we can decide if $(x, k) \in L$ in running time $f(k) \cdot |x|^{\mathcal{O}(1)}$, and $f$ is an arbitrary function. Thus, when studying an NP-hard problem, we are interested in finding a parameter $k$ such that the problem lies in FPT when parameterized by $k$. As a last recourse when facing NP-hard problems, we also consider XP algorithms, where for a parameter $k$ we are allowed to solve the problem in running time $n^{f(k)}$ where $n$ is the size of the input.

# Part I

# Vertex Splitting

CHAPTER 3

# General Graph Drawings

While planar graphs admit compact and naturally crossing-free drawings, most graphs that we encounter regularly have no guarantee to be planar. Computing good layouts of large and dense non-planar graphs remains a challenging task, mainly due to the visual clutter caused by large numbers of edge crossings. Several methods have been proposed to simplify the drawings of non-planar graphs and to minimize crossings, both from a practical point of view [LHT17, vLKS+11] and a theoretical one [Lie01, Sch18]. Drawing algorithms often focus on reducing the number of visible crossings [Nö20] or improving crossing angles [Oka20], aiming to achieve similar beneficial readability properties as in crossing-free drawings of planar graphs. In this chapter, we begin our investigation of the vertex splitting operation applied to graph drawings by focusing on the problem of transforming non-plane drawings of graphs into plane ones with vertex splitting. This chapter is based on joint work with Martin Nöllenburg, Manuel Sorgue, Soeren Terziadis, Jules Wulms and Hsiang-Yun Wu [NST+22]. This paper was presented at GD 2022, and is currently submitted to JoCG. An earlier version was presented at EuroCG 2022.

Since planarity is a highly desirable property when dealing with graph drawing, we first try to map out the different problems that are related to splitting vertices and obtaining planar drawings, and then try to understand their complexity, to hopefully derive efficient algorithms for vertex splitting. Recall a *vertex split* operation on the vertex $v$: a vertex split removes $v$ from $G$ and instead adds two non-adjacent copies $v_1, v_2$ such that the edges formerly incident to $v$ are distributed among $v_1$ and $v_2$. Similarly, a *$k$-split* of $v$ for $k \geq 2$ creates $k$ copies $v_1, \ldots, v_k$, among which the edges formerly incident to $v$ are distributed. On the one hand, splitting a vertex can resolve some of the crossings of its incident edges, but on the other hand the number of objects in the drawing to keep track of increases. Therefore, we aim to minimize the number of splits needed to obtain a planar graph, which is known as the *splitting number* of the graph. A related concept is the planar split thickness [EKK+18] of a graph $G$, which is the minimum $k$ such that $G$ can be decomposed into at most $k$ planar subgraphs by applying a $k$-split to each vertex

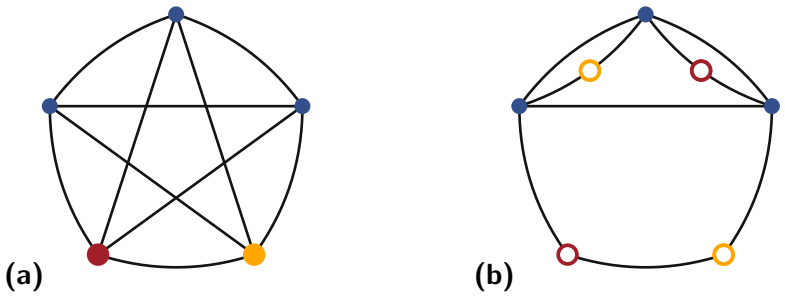



Figure 3.1: Vertex splitting in a drawing of $K_5$. The red and orange disks in **(a)** are split once into the red and orange circles in **(b)**. Note that an abstract $K_5$ without drawing has splitting number 1.

of $G$ at most once. Eppstein et al. [EKK+18] showed that deciding whether a graph has split thickness $k$ is NP-complete, even for $k = 2$, but can be approximated within a constant factor and is fixed-parameter tractable (FPT) for graphs of bounded treewidth.

While previous work considered vertex splitting in the context of abstract graphs, here we want to improve a given input drawing by splitting a minimum number of vertices, which can be freely re-embedded, while the non-split vertices must remain at their original positions in order to maintain layout stability [MELS95], see Fig. 3.1.

As introduced in Chapter 1, the underlying algorithmic problem for vertex splitting in drawings of graphs poses many challenges. Here we separate the problem in the following manner: firstly, a suitable (minimum) subset of vertices to be split must be selected, and secondly the split copies of these vertices must be re-embedded in a crossing-free way. The original edges of each split vertex are partitioned into a subset for each of its copies.

Similarly to the vertex explosion graph operation introduced in Chapter 1, the former problem of selecting split vertices is closely related to the NP-complete problem Vertex Deletion (sometimes called Vertex Planarization), where we want to decide whether a given graph can be made planar by deleting at most $k$ vertices, and to related problems of hitting graph minors by vertex deletions, and is very well studied in the parameterized complexity realm [JLS13, Kaw09, KLP+15, RS95]. For example, it follows from results of Robertson and Seymour [RS95] that Vertex Deletion can be solved in cubic time for fixed $k$. A series of papers [JLS13, Kaw09, MS12] improved the dependency on the input size to linear and the dependency on $k$ to $2^{O(k \log k)}$.

The latter re-embedding problem is not only related to drawing extension problems, but as we show in Section 3.3, it generalizes natural problems on covering vertices by faces in planar graphs [AFL08, AL04, BM88, KLL02].

**Structure of the chapter:** In Section 3.2 we start by providing a non-uniform FPT algorithm for the original (non-embedded) splitting number problem. We show that the class of graphs that can be made planar by at most $k$ splitting operations is closed

under taking minors. Additionally, we provide a polynomial time algorithm that, when given a drawing, a vertex, and an integer $k$, can compute a crossing minimal drawing of the input graph when splitting the input vertex $k$ times. In Section 3.3, we show that both subproblems that were outlined above are NP-complete, as well as the overall problem. Lastly, in Section 3.4, we present an FPT algorithm that solves the re-embedding subproblem, by reducing our instance to a bounded outerplanarity case that allows us to use a dynamic program on a sphere-cut decomposition.

## 3.1 Preliminaries

Let $G = (V, E)$ be a simple graph with vertex set $V(G) = V$ and edge set $E(G) = E$. For a subset $V' \subset V$, $G[V']$ denotes the subgraph of $G$ induced by $V'$. The neighborhood of a vertex $v \in V(G)$ is defined as $N_G(v)$. If $G$ is clear from the context, we omit the subscript $G$. A *split* operation applied to a vertex $v$ results in a graph $G' = (V', E')$ where $V' = V \setminus \{v\} \cup \{\dot{v}^{(1)}, \dot{v}^{(2)}\}$ and $E'$ is obtained from $E$ by distributing the edges incident to $v$ among $\dot{v}^{(1)}, \dot{v}^{(2)}$ such that $N_G(v) = N_{G'}(\dot{v}^{(1)}) \cup N_{G'}(\dot{v}^{(2)})$ (copies are written with a dot for clarity). Splits with $N(\dot{v}^{(1)}) = N(v)$ and $N(\dot{v}^{(2)}) = \emptyset$ (equivalent to moving $v$ to $\dot{v}^{(1)}$), or with $N(\dot{v}^{(1)}) \cap N(\dot{v}^{(2)}) \neq \emptyset$ (which is never beneficial, but can simplify proofs) are allowed. The vertices $\dot{v}^{(1)}, \dot{v}^{(2)}$ are called *split vertices* or *copies* of $v$. If a copy $\dot{v}$ of a vertex $v$ is split again, then any copy of $\dot{v}$ is also called a copy of the original vertex $v$ and we use the notation $\dot{v}^{(i)}$ for $i = 1, 2, \ldots$ to denote the different copies of $v$.

**Problem 1** (Splitting Number)**.** Given a graph $G = (V, E)$ and an integer $k$, can $G$ be transformed into a planar graph $G'$ by applying at most $k$ splits to $G$?

Splitting Number is NP-complete, even for cubic graphs [FdFdMN01]. We extend the notion of vertex splitting to drawings of graphs. Let $G$ be a graph and let $\Gamma$ be a *topological drawing* of $G$, which maps each vertex to a point in $\mathbb{R}^2$ and each edge to a simple curve connecting the points corresponding to the incident vertices of that edge. We still refer to the points and curves as vertices and edges, respectively, in such a drawing. Furthermore, we assume $\Gamma$ is a *simple* drawing, meaning no two edges intersect more than once, no three edges intersect in one point (except common endpoints), and adjacent edges do not cross.

**Problem 2** (Embedded Splitting Number)**.** Given a graph $G = (V, E)$ with a simple topological drawing $\Gamma$ and an integer $k$, can $G$ be transformed into a graph $G'$ by applying at most $k$ splits to $G$ such that $G'$ has a planar drawing that coincides with $\Gamma$ on $G[V(G) \cap V(G')]$?

Problem 2 includes two interesting subproblems, namely an embedded vertex deletion problem (which corresponds to selecting candidates for splitting) and a subsequent re-embedding problem, both defined below.

**Problem 3** (Embedded Vertex Deletion)**.** Given a graph $G = (V, E)$ with a simple topological drawing $\Gamma$ and an integer $k$, can we find a set $S \subset V$ of at most $k$ vertices such that the drawing $\Gamma$ restricted to $G[V \setminus S]$ is planar?

Problem 3 is closely related to the NP-complete problem Vertex Deletion [JLS13, Kaw09, MS12], yet it deals with deleting vertices from an arbitrary given drawing of a graph with crossings. One can easily see that Problem 3 is FPT, using a bounded search tree approach, where for up to $k$ times we select a remaining crossing and branch over the four possibilities of deleting a vertex incident to the crossing edges. The vertices split in a solution of Problem 2 necessarily are a solution to Problem 3; otherwise some crossings would remain in $\Gamma$ after splitting and re-embedding. However, a set corresponding to a minimum-split solution of Problem 2 is not necessarily a minimum cardinality vertex deletion set as vertices can be split multiple times. Moreover, an optimal solution to Problem 2 may also split vertices that are not incident to any crossed edge and thus do not belong to an inclusion-minimal vertex deletion set. We note here that a solution to Problem 3 solves a problem variation where rather than minimizing the number of splits required to reach planarity, we instead minimize the number of split vertices: Splitting each vertex in an inclusion-minimal vertex deletion set its degree many times trivially results in a planar graph.

In the re-embedding problem, we are given a graph drawing and a set of candidate vertices to be split. The task is to decide how many times to split each candidate vertex, where to re-embed each copy, and to which neighbors of the original candidate vertex to connect each copy.

**Problem 4** (Split Set Re-Embedding)**.** Given a graph $G = (V, E)$, a candidate set $S \subset V$ such that $G[V \setminus S]$ is planar, a simple planar topological drawing $\Gamma$ of $G[V \setminus S]$, and an integer $k \geq |S|$, can we perform in $G$ at most $k$ splits to the vertices in $S$, where each vertex in $S$ is split at least once, such that the resulting graph has a planar drawing that coincides with $\Gamma$ on $G[V \setminus S]$?

We note that, if no splits were allowed ($k = 0$), then Problem 4 would reduce to a partial planar drawing extension problem asking to re-embed each vertex of set $S$ at a new position without splitting, which can be solved in linear time [ADBF+15].

## 3.2 Algorithms for (Embedded) Splitting Number

Splitting Number is known to be NP-complete in non-embedded graphs [FdFdMN01]. In Section 3.2.1, we show that it is FPT when parameterized by the number of allowed split operations. Indeed, we will show something more general, namely, that we can replace planar graphs by any class of graphs that is closed under taking minors and still get an FPT algorithm. Essentially we will show that the class of graphs that can be made planar by at most $k$ splitting operations is closed under taking minors and then apply a

result of Robertson and Seymour that asserts that membership in such a class can be checked efficiently [CFK+15b].

For vertex splitting in graph drawings, we consider in Section 3.2.2 the restricted problem to split a single vertex. We show that selecting such a vertex and re-embedding at most $k$ copies of it, while minimizing the number of crossings, can be done in polynomial time for constant $k$.

### 3.2.1 A Non-Uniform Algorithm for Splitting Number

We use the following terminology. A *minor* of a graph $G$ is a graph $H$ obtained from a subgraph of $G$ by a series of edge contractions. *Contracting* an edge $uv$ means to remove $u$ and $v$ from the graph, and to add a vertex that is adjacent to all previous neighbors of $u$ and $v$. A graph class $\Pi$ is *minor closed* if for every graph $G \in \Pi$ and each minor $H$ of $G$ we have $H \in \Pi$. Let $\Pi$ be a graph class and $k \in \mathbb{N}$. We define the graph class $\Pi_k$ to contain each graph $G$ such that a graph in $\Pi$ can be obtained from $G$ by at most $k$ vertex splits.

**Theorem 1.** *For a minor-closed graph class $\Pi$ and $k \in \mathbb{N}$, $\Pi_k$ is minor closed.*

*Proof.* Let $G \in \Pi_k$ and let $H$ be a minor of $G$. We show that $H \in \Pi_k$. Let $G'$ be a subgraph of $G$ such that $H$ is obtained from $G'$ by a series of edge contractions. We first show that $G' \in \Pi_k$. Let $s_1, s_2, \ldots, s_{k'}$ be a sequence of at most $k$ vertex splits that, when successively applied to $G$, we obtain a graph in $\Pi$. Let $G_0 = G$ and for each $i = 1, \ldots, k'$ let $G_i$ be the graph obtained after applying $s_i$. We adapt the sequence $s_1, \ldots, s_{k'}$ to obtain a sequence of graphs $G' = G'_0, G'_1, \ldots, G'_{k'}$ as follows. For each $i = 1, 2, \ldots, k'$, if $s_i$ is applied to a vertex $v \in V(G_{i-1})$ with partition $N_1, N_2$ of $N_{G_{i-1}}(v)$ then, if $v \in V(G'_{i-1})$, to get $G'_i$ we apply a split operation in $G'_{i-1}$ to $v$ with neighborhood cover $(N_1 \cap V(G'_{i-1}), N_2 \cap V(G'_{i-1}))$ (and we assume without loss of generality that the new vertices introduced by this operation are identical to the vertices introduced by $s_i$). If $v \notin V(G'_{i-1})$ we put $G_i = G_{i-1}$ instead. Observe that for each $i \in \{1, 2, \ldots, k'\}$, graph $G'_i$ is a subgraph of $G_i$. Hence, $G'_{k'}$ is a subgraph of $G_{k'}$ and since $\Pi$ is in particular closed under taking subgraphs, we have $G'_{k'} \in \Pi$, as claimed.

It remains to show that for each graph $H$ that is obtained from a graph $G \in \Pi_k$ through a series of edge contractions we have $H \in \Pi_k$. By induction on the number of edge contractions, it is enough to show this in the restricted case where $H$ is obtained by a single edge contraction from $G$. Let $s_i$ and $G_i$ be as defined above, that is, $s_1, \ldots, s_{k'}$ is a sequence of vertex-split operations that when successively applied to $G$, we obtain a graph in $\Pi$ and $G_i$ is the graph obtained after applying $s_i$. We claim that there is a series of $k'$ split operations applied to $H = H_0$ that result in a series of graphs $H_1, H_2, \ldots, H_{k'}$ such that for all $i \in \{1, 2, \ldots, k'\}$ a graph isomorphic to $H_i$ is obtained from $G_i$ by a single edge contraction. This would imply that $H_{k'} \in \Pi$ because $\Pi$ is minor closed and thus we would have $H \in \Pi_k$, finishing the proof.

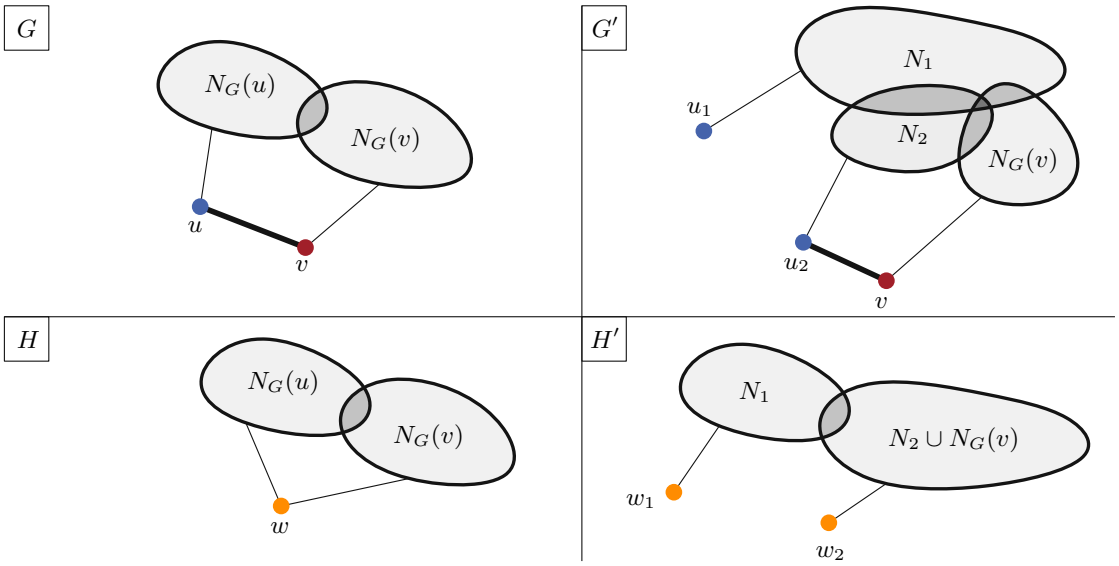



Figure 3.2: Graph $G$ in which splitting vertex $u$ results in graph $G'$ (top row), and minor $H$, the result of contracting edge $uv$ in $G$, where splitting $w$ results in minor $H'$ of $G'$, by contracting $u_2v$ (bottom row). The set $N_G(u)$ is split into $N_1$ and $N_2$. An edge between a vertex and a set indicates the possibility for the vertex to have neighbors in that set.

To prove the claim, since $H_0$ is obtained from $G_0$ by a single edge contraction, by induction it is enough to show the following. Let $H = (V', E')$ be obtained from $G = (V, E)$ by contracting the single edge $uv \in E$ and let $w \in V'$ be the vertex resulting from the contraction. Let $G'$ be obtained from $G$ by applying a single split operation $s$. It is enough to show that there is a split operation such that, when applied to $H$ to obtain the graph $H'$ we have that $H'$ is isomorphic to a graph obtained from $G'$ by contracting a single edge.

To show this, if $s$ is not applied to $u$ or $v$, then $s$ can directly be applied to $H$. Then, contracting $uv$ in $G'$ we see directly that we obtain a graph isomorphic to $H'$, as required.

Otherwise, $s$ is applied to $u$ or $v$. By symmetry, say $s$ is applied to $u$ without loss of generality. Let thus $u \in V(G)$ be split into $u_1$ and $u_2$ in $G'$. We split $w$ into $w_1$ and $w_2$ in $H$ with the neighborhood 2-cover of $w$ defined as $(N_{G'}(u_1), N_{G'}(u_2) \cup N_{G'}(v))$. That is, $H'$ is obtained from $H$ by splitting $w$ into $w_1$ and $w_2$ such that $N_{H'}(w_1) = N_{G'}(u_1)$ and $N_{H'}(w_2) = N_{G'}(u_2) \cup N_{G'}(v)$ (see Fig. 3.2). Contracting $u_2v$ in $G'$ into a vertex $x$ results in a graph $\hat{G}$ that is isomorphic to $H'$: To see this, observe that all neighborhoods of vertices in $V \setminus \{u_1, x\}$ are identical between $\hat{G}$ and $H'$ and that $N_{\hat{G}}(u_1) = N_{H'}(w_1)$ and $N_{\hat{G}}(x) = N_{G'}(u_1) \cup N_{G'}(v) = N_{H'}(w_2)$. Thus, our claim is proven. □

By results of Robertson and Seymour we obtain:

**Proposition 1.** Let $\Pi$ be a minor-closed graph class. There is a function $f\colon \mathbb{N} \to \mathbb{N}$ such that for every $k \in \mathbb{N}$ there is an algorithm running in $f(k) \cdot n^3$ time that, given a graph $G$ with $n$ vertices, correctly determines whether $G \in \Pi_k$.

*Proof.* From Theorem 1 it follows that the class $\Pi_k$ of graphs that represent positive input instances is closed under taking minors. From Robertson and Seymour's graph minor theorem it follows that it can be determined in $c_{\Pi_k} \cdot n^3$ time for a given $n$-vertex graph whether it is contained in $\Pi_k$, where $c_{\Pi_k}$ is a constant depending only on $\Pi_k$ (see Cygan et al. [CFK+15b, Theorem 6.13]). Proposition 1 follows by setting $f(k) = c_{\Pi_k}$. □

Since the class of planar graphs is minor closed, we obtain the following.

**Corollary 1.** Splitting Number is non-uniformly fixed-parameter tractable[1] with respect to the number of allowed vertex splits.

### 3.2.2 Optimally Splitting a Single Vertex in a Graph Drawing

Given a graph $G = (V, E)$, its drawing $\Gamma$, a candidate vertex $v \in V$ and an integer $k$, we show that we can split $v$ into $k$ copies such that the resulting number of crossings is minimized; we construct a corresponding crossing-minimal drawing $\Gamma^*$ in polynomial time.

Chimani et al. [CGMW09] showed that inserting a single star into an embedded graph while minimizing the number of crossings can be solved in polynomial time. They use the fact that the length of a shortest path in the dual graph between two vertices $v_1$ and $v_2$ corresponds to the number of edge crossings generated by an edge between two vertices embedded on the faces in the primal graph corresponding to $v_1$ and $v_2$. We extend this method to optimally split a single vertex into $k$ copies, which is similar to reinserting $k$ stars. The algorithm planarizes the input graph, then exhaustively tries all combinations of $k$ faces to re-embed the copies of $v$. For each combination it finds for all the neighbors of $v$ which copy is their best new neighbor (meaning it induces the least number of crossings) using the dual graph.

In the first step we remove the specified vertex $v$ from $\Gamma$ with all its incident edges. Let $\Pi$ be the planarization of $\Gamma \setminus v$, let $F$ be the set of faces of $\Pi$, and let $D$ be the dual graph of $\Pi$. For a vertex $u \in \Pi$ incident to a face set $F(u)$, we define $V_F(u)$ as the vertex set that represents $F(u)$ in $D$. The algorithm by Chimani et al. [CGMW09] computes the crossing number for the vertex insertion in a face $f \in F$ by finding the sum of the shortest paths in $D$ between the vertex $v_f$ that represents $f$ in $D$ and $V_F(w)$, for each $w \in N(v)$. In our algorithm, we collect all the individual path lengths in $D$ between each face vertex $v_f$ and the faces in $V_F(w)$, for each $w \in N(v)$, in a table, and then consider all face subsets of size $k$ in which to embed the $k$ copies of $v$. For such a given

[1] A parameterized problem is non-uniformly fixed-parameter tractable if there is a function $f\colon \mathbb{N} \to \mathbb{N}$ and a constant $c$ such that for every parameter value $k$ there is an algorithm that decides the problem and runs in $f(k) \cdot n^c$ time on inputs with parameter value $k$ and length $n$.

set $S$ of $k$ faces, we assign each $w \in N(v)$ to its closest face $f^*(w) \in S$, which yields a crossing-minimal edge between $w$ and $S$. We break ties in path lengths lexicographically using a fixed order of $F$. For each set $S$ of candidate faces we compute the sum of the resulting shortest path lengths between $w$ via some face in $V_F(w)$ and $f^*(w)$ for each $w \in N(v)$.

We choose as the solution the set $S^*$ with minimum total path length and assign one copy $\dot{v}^{(1)}, \ldots, \dot{v}^{(k)}$ of $v$ into each face of $S^*$ that is closest to at least one of the neighbors in $N(v)$. This corresponds to splitting $v$ into at most $k$ copies. The edges from the newly placed copies to the neighbors of $v$ follow the computed shortest paths in $D$.

Chimani et al. [CGMW09] showed that we can compute the table of path lengths in $D$ in time $O((|F| + |\mathcal{E}|)|N(v)|)$, where $F$ and $\mathcal{E}$ are, respectively, the sets of faces and edges of $\Pi$. We consider $O(|F|^k)$ subsets of $k$ faces and chose the one minimizing crossings. Thus our algorithm runs in polynomial time for $k \in O(1)$.

**Theorem 2.** *Given a drawing $\Gamma$ of a graph $G$, a vertex $v \in V(G)$, and an integer $k$, we can split $v$ into $k$ copies such that the remaining number of crossings is minimized in time $O((|F| + |\mathcal{E}|) \cdot |N(v)| \cdot |F|^k)$, where $F$ and $\mathcal{E}$ are respectively the sets of faces and edges of the planarization of $\Gamma$.*

*Proof.* Given a solution that embeds copies in the faces $f_1, \ldots, f_k$, the drawing we compute has a minimum number of edge crossings. Since for each face combination the algorithm finds the minimum number of crossings with the edges of $\Gamma$ of a vertex insertion, by design, the star centered at each copy has minimum number of crossings with $\Gamma$ by the exhaustive search. Still, uncounted crossings between inserted edges could happen. We argue that this is impossible. Let us assume that we have two stars rooted on $v_i$, $v_j$, and vertices $u_i, u_j \in N(v)$ s.t. $(v_i, u_i)$ crosses the edge $(v_j, u_j)$ in a point $c$ inside some face $f$. This means that in the dual graph both paths representing the two edges pass through the same vertex $v_f$ that represents $f$. If the path between $c$ and $v_i$ crosses less edges than the path between $c$ and $v_j$ then the path in $D$ from $v_j$ to $u_j$ is not minimal and $u_j$ would have been assigned to $v_i$ (symmetrically if the path between $c$ and $v_j$ crosses less edges). If both paths have the same length, then by the lexicographic order rule, both vertices would have been assigned to the same face. So the stars do not intersect one another and we search exhaustively for all possible face sets to embed the stars, meaning that the algorithm produces a crossing minimal drawing after splitting $v$ at most $k$ times. Finally, to find the number of crossings generated by adding an edge between a neighbor $w$ of $v$, we can do a BFS traversal of the graph from the faces of $V_F(w)$, and every time we encounter a face, the depth on the tree corresponds to the face distance to $V_F(w)$. We do this for every element of $N(v)$, and for every set of $k$ faces of the planarization of $\Gamma$, which takes $O((|F| + |\mathcal{E}|) \cdot |N(v)| \cdot |F|^k)$ time. □

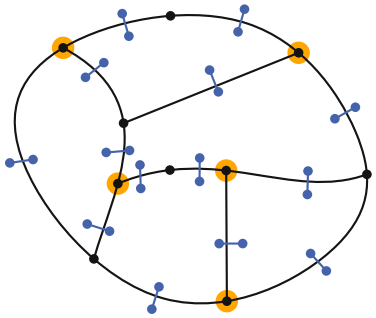


Figure 3.3: The drawing $\Gamma$ in black, and the vertices and edges added to obtain $\Gamma'$ in blue. The vertex cover highlighted in orange corresponds to the deletion set.

## 3.3 NP-completeness of Subproblems

While it is known that Splitting Number is NP-complete [FdFdMN01], in the correctness proof of the reduction Faria et al. [FdFdMN01] assume that it is permissible to draw all vertices, split or not, at new positions as there is no initial drawing to be preserved. The reduction thus does not seem to easily extend to Embedded Splitting Number. Here we show the NP-completeness of each of its two subproblems.

**Theorem 3.** *Embedded Vertex Deletion is NP-complete.*

*Proof.* We reduce from the NP-complete Vertex Cover problem in planar graphs [GJ77], where given a planar graph $G = (V, E)$ and an integer $k$, the task is to decide if there is a subset $V' \subseteq V$ with $|V'| \leq k$ such that each edge $e \in E$ has an endpoint in $V'$. Given the planar graph $G$ from such a Vertex Cover instance and an arbitrary plane drawing $\Gamma$ of $G$ we construct an instance of Embedded Vertex Deletion as follows. We create a drawing $\Gamma'$ by drawing a *crossing edge* $e'$ across each edge $e$ of $\Gamma$ such that $e'$ is orthogonal to $e$ and has a small enough positive length such that $e'$ intersects only $e$ and no other crossing edge or edge in $\Gamma$, see Fig. 3.3. Drawing $\Gamma'$ can be computed in polynomial time.

Let $C$ be a vertex cover of $G$ with $|C| = k$. We claim that $C$ is also a deletion set that solves Embedded Vertex Deletion for $\Gamma'$. We remove the vertices in $C$ from $\Gamma'$, with their incident edges. By definition of a vertex cover, this removes all the edges of $G$ from $\Gamma'$. The remaining edges in $\Gamma'$ are the crossing edges and they form together a (disconnected) planar drawing which shows that $C$ is a solution of Embedded Vertex Deletion for $\Gamma'$.

Let $D$ be a deletion set of $\Gamma'$ such that $|D| = k$. We find a vertex cover of size at most $k$ for $G$ in the following manner. Assume that $D$ contains a vertex $w$ that is an endpoint of a crossing edge $e$ that crosses the edge $(u, v)$ of $G$. Since $w$ has degree one, deleting it only resolves the crossing between $e$ and $(u, v)$, thus we can replace $w$ in $D$ by $u$ (or $v$) and resolve the same crossing as well as all the crossings induced by the edges incident to $u$ (or $v$). Thus we can find a deletion set $D'$ of size smaller or equal to $k$ that contains

only vertices in $G$ and removing this deletion set from $\Gamma'$ removes only edges from $G$. Since every edge of $G$ is crossed in $\Gamma'$, every edge of $G$ must have an incident vertex in $D'$, thus $D'$ is a vertex cover for $G$.

Containment in NP is easy to see. Given a deletion set $D$, we only need to verify that $\Gamma'$ is planar after deleting $D$ and its incident edges. □

Next, we prove that also the re-embedding subproblem itself is NP-complete, by showing that Face Cover is a special case of the re-embedding problem. The problem Face Cover is defined as follows. Given a planar graph $G = (V, E)$, a subset $D \subseteq V$, and an integer $k$, can $G$ be embedded in the plane, such that at most $k$ faces are required to cover all the vertices in $D$? Face Cover is NP-complete, even when $G$ has a unique planar embedding [BM88].

**Theorem 4.** *Split Set Re-Embedding is NP-complete.*

*Proof.* We give a parameterized reduction from Face Cover (with unique planar embedding) parameterized by the solution size to the re-embedding problem parameterized by the number of allowed splits. We first create a graph $G' = (V', E')$, with a new vertex $v$, vertex set $V' = V \cup \{v\}$ and $E' = E \cup \{dv \mid d \in D\}$. Then we compute a planar drawing $\Gamma$ of $G$ corresponding to the unique embedding of $G$. Finally, we define the candidate set $S = \{v\}$ and allow for $k - 1$ splits in order to create up to $k$ copies of $v$. Then $G'$, $\Gamma$, $S$, and $k - 1$ form an instance $I$ of Split Set Re-Embedding.

In a solution of $I$, every vertex in $D$ is incident to a face in $\Gamma$, in which a copy of $v$ was placed. Therefore, selecting these at most $k$ faces in $\Gamma$ gives a solution for the Face Cover instance. Conversely, given a solution for the Face Cover instance, we know that every vertex in $D$ is incident to at least one of the at most $k$ faces. Therefore, placing a copy of $v$ in every face of the Face Cover solution yields a re-embedding of at most $k$ copies of $v$, each of which can realize crossing-free edges to all its neighbors incident to the face.

Finally, a planar embedding of the graph can be represented combinatorially in polynomial space. We can also verify in polynomial time that this embedding is planar and exactly the right connections are realized, for NP-containment. □

## 3.4 Split Set Re-Embedding is Fixed-Parameter Tractable

In this section we propose an FPT algorithm for Problem 4 (Split Set Re-Embedding) and prove the following theorem.

**Theorem 5.** *Split Set Re-Embedding can be solved in $2^{O(k^2)} \cdot n^{O(1)}$ time, where $k$ is the number of allowed splits and $n$ is the number of vertices in the input graph $G$.*

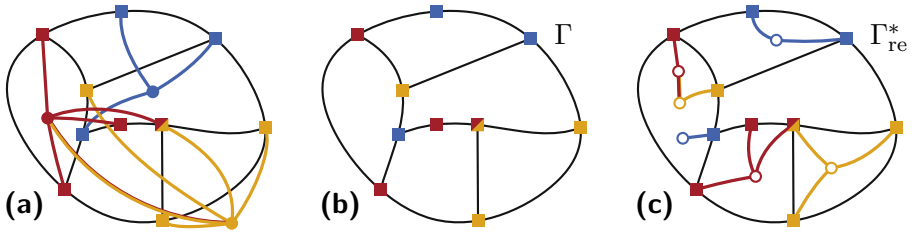



Figure 3.4: **(a)** An example graph $G$, **(b)** a planar drawing $\Gamma$ of $G$ where $S$ has been removed, and **(c)** a solution drawing $\Gamma^*_{\text{re}}$. Pistils are squares, copies are circles and vertices in $S$ are disks.

**Outline of Section 3.4.** An overview of our algorithm for Theorem 5 is as follows. Recall that we are given an instance $(G, \Gamma, S, k)$ consisting of a graph $G = (V, E)$, a set $S$ of candidate vertices, a drawing $\Gamma$ of $G[V \setminus S]$, and an integer $k$. We aim to split the candidate vertices at most $k$ times and to re-embed them in the simple drawing $\Gamma$ using dynamic programming with the following setup. First, from the given set $S$ of $s = |S|$ candidate vertices (disks in Fig. 3.4(a)) we guess, by trying all possible combinations, how many copies of each candidate we will insert back into the graph and we determine the connections among the copies. We then remove unimportant elements of the drawing that are too far away from the faces relevant for re-inserting vertices and edges. As we will see, this results in a $10k$-outerplanar graph, the details can be found in Section 3.4.1. Next, we remove all bridges, which allows us to compute a so-called sphere-cut decomposition of the drawing (Section 3.4.2) over which we will later perform dynamic programming.

The dynamic-programming algorithm will maintain tables that keep track of possible partial solutions. We introduce partial solutions formally in Section 3.4.3 together with their succinct encoding as so-called *signatures* we use to describe them as well as their properties. Finally, this allows us to present our dynamic-programming algorithm in Section 3.4.4. In the base case, the algorithm looks at a leaf of the tree, which represents an edge $e$ of the drawing $\Gamma$, and embeds copies of split vertices in the two faces on either side of $e$. In the general case, it combines two sub-solutions (vertex embeddings) that are *consistent*. Consistent sub-solutions jointly do not use more copies than allowed by parameter $k$, and they can be combined in a planar way.

To aid the explanations in the upcoming sections, we first introduce the following terminology. For an instance $(G, \Gamma, S, k)$ of Split Set Re-Embedding, any vertex $v$ in $\Gamma$, the input drawing, that has a neighbor in candidate set $S$, the set of vertices to be split, is called a *pistil*. Each face of $\Gamma$ that is incident to a pistil is called a *petal*. Let $p$ be a pistil in the input graph $G$ with neighbors $N(p)$. Let $\dot{v}$ be a copy of some $v \in S$, where $v \in N(p)$. We say $\dot{v}$ *covers* $p$ if it is embedded in a face incident to $p$ and we can draw a crossing-free edge between $\dot{v}$ and $p$.

If $(G, \Gamma, S, k)$ is a yes-instance of Split Set Re-Embedding, then there is a series of split operations to achieve a planar re-embedding. We call the elements of this series *solution splits* (see Fig. 3.4 for an example of the problem and its solutions), and we refer

to the *solution graph* as the graph obtained from $G$ by performing the solution splits. Formally, a *solution* is defined as a tuple $(S^*_\curlywedge, \mathsf{copies}^*, \mathsf{orig}^*, (N^*_u)_{u\in S^*_\curlywedge}, \Gamma^*_{\mathrm{re}})$ consisting of the following:

i The set $S^*_\curlywedge$ of copies of vertices in $S$ introduced by the solution splits. Since $S$ has size $s \leq k$ and there are at most $k$ splits, we have $|S^*_\curlywedge| \leq s + k \leq 2k$.

ii A mapping $\mathsf{copies}^*\colon S \to 2^{S^*_\curlywedge}$ that maps each vertex $v \in S$ to the set $\mathsf{copies}^*(v) \subseteq S^*_\curlywedge$ of copies of $v$ introduced by performing the solution splits.

iii A mapping $\mathsf{orig}^*\colon S^*_\curlywedge \to S$ that maps each copy $\dot{v} \in S^*_\curlywedge$ to the vertex $v = \mathsf{orig}^*(\dot{v}) \in S$ that $\dot{v}$ is a copy of.

iv For each copy $\dot{v} \in S^*_\curlywedge$ a vertex set $N^*_{\dot{v}} \subseteq V$ of neighbors of $\dot{v}$ such that for each $v \in S$ the family $\{N^*_{\dot{v}} \mid \dot{v} \in \mathsf{copies}^*(v)\}$ is a partition of $N_G(v)$.

v A planar drawing $\Gamma^*_{\mathrm{re}}$ of the graph resulting from $\Gamma$ by embedding the copies in $S^*_\curlywedge$ such that each copy $\dot{v} \in S^*_\curlywedge$ has edges drawn to each vertex in $N^*_{\dot{v}}$ (Fig. 3.4c).

### 3.4.1 Splitting Candidates and Obtaining $10k$-Outerplanarity.

**Initializing the Split Vertex Set $S_\curlywedge$** As mentioned, the first step in our algorithm for Split Set Re-Embedding is to determine, for each candidate split vertex, how many copies are introduced and how these copies are connected to each other. We are given as input the set $S$ of vertices that have been removed from the drawing to be split such that $|S| = s \leq k$. Otherwise, if $|S| = s > k$, we can immediately conclude that we deal with a no-instance as all candidate vertices must be split at least once. From $S$ we now create sets $S_\curlywedge$ of copies that will be reintroduced into the drawing. To obtain all relevant sets $S_\curlywedge$, we introduce our first branching rule.

**Branching Rule 1.** *Let $(G, \Gamma, S, k)$ be an instance of Split Set Re-Embedding. Create a branch for every $k' = s, \ldots, s+k$, defining a set $S_\curlywedge$ of $k'$ new vertices, for every mapping $\mathsf{orig}\colon S_\curlywedge \to S$, and every mapping $\mathsf{copies}\colon S \to 2^{S_\curlywedge}$ such that $\{\mathsf{copies}(v) \mid v \in S\}$ is a partition of $S_\curlywedge$ and such that $\mathsf{orig}^{-1} = \mathsf{copies}$.*[2]

Note that for resolving crossings in $\Gamma$ one might sometimes want to only re-embed a vertex without splitting. However, such a vertex move operation is not permitted in Embedded Splitting Number and we only permit re-embedding copies of original vertices. Thus each vertex in $S$ is split at least once. We can bound the number of branches by observing that the number of ways of distributing the $k$ splits among $s$ vertices with at least one split per vertex is the same as the number of $s$-compositions of the integer $k$, which is described by $\binom{k-1}{s-1} \leq 2^k$.

[2]Herein for a mapping $f\colon X \to Y$ we define the mapping $f^{-1}\colon Y \to 2^X$ by putting $f^{-1}(y) = \{x \in X \mid f(x) = y\}$ for every $y \in Y$.

**Lemma 1.** *If $(G, \Gamma, S, k)$ is a yes-instance of* Split Set Re-Embedding *with solution $\sigma = (S^*_\curlywedge, \mathit{copies}^*, \mathit{orig}^*, (N_{\dot{v}})_{\dot{v} \in S^*_\curlywedge}, \Gamma^*_{\mathrm{re}})$, then there is a branch created by Branching Rule 1 with $S^*_\curlywedge = S_\curlywedge$.*

*Proof.* Given the set $S^*_\curlywedge$ of copies re-embedded by a solution, we argue that $S^*_\curlywedge$ must be a subset of a set generated in at least one of the branches of BR1. Set $S^*_\curlywedge$ necessarily contains at least one copy of each vertex in $S$, and we call $S_{\mathrm{add}}$ the remainder of copies embedded by $S^*_\curlywedge$. Since the total number of splits is bounded by $k$, $S_{\mathrm{add}}$ contains at most $k = s + d$ vertices: one copy of each vertex in $S$ for the initial split, and $d$ vertices for additional splits. Our branching rule BR1 creates a branch for each combination of copies from $S$ that adds up to $2s + d$, two copies of each vertex in $S$, and one branch for each combination of $d$ copies of $S$. Thus there is a branch $b$ that chooses a set of copies $S_b$ that contains a superset of $S_{\mathrm{add}}$, such that $S_b \setminus S_{\mathrm{add}}$ contains exactly one copy of each vertex in $S$. Thus $S_b$ contains exactly all copies $S^*_\curlywedge$ re-embedded by a solution. □

Some vertices in $S_\curlywedge$ have neighbors in $S_\curlywedge$, or in both $S_\curlywedge$ and $V \setminus S$. From the perspective of a non-split pistil it is easy to verify whether the original neighborhood is present in a solution. However, a split vertex is incident to only a subset of its original edges. Thus it is required to consider the union of all copies of a vertex to find its original neighborhood. For an edge $(u, v)$ where $u, v \in S$, we will branch over all possibilities for defining this edge between any pair of copies of $u$ and $v$. So if there are $i$ copies of $u$ and $j$ copies of $v$, we will branch over the $i \cdot j$ possible ways to reflect the edge $(u, v)$ in the solution.

**Branching Rule 2.** *Let $(G = (V, E), \Gamma, S, k)$ be an instance of* Split Set Re-Embedding *and $S_\curlywedge$ a set of copies obtained from Branching Rule 1. Create a branch for each possible set $E_\curlywedge \subseteq \{(\dot{u}^{(i)}, \dot{v}^{(j)}) \mid \dot{u}^{(i)}, \dot{v}^{(j)} \in S_\curlywedge \text{ and } (\mathit{orig}^*(\dot{u}^{(i)}), \mathit{orig}^*(\dot{v}^{(j)})) \in E\}$ such that every $e \in E$ is represented exactly once in $E_\curlywedge$. That is, for each $(u, v) \in {S_\curlywedge}^2 \cap E$ there is exactly one $i$ and exactly one $j$ such that $(\dot{u}^i, \dot{v}^j) \in E_\curlywedge$.*

Let the number $s$ of splits satisfy $s \geq 2$ as otherwise there are no edges with both endpoints in $S_\curlywedge$. Each of the $s \leq k$ vertices in $S$ has at most $k - 1$ neighbors in $S$, yielding at most $\binom{k}{2}$ edges. Since each vertex in $S$ has at most $k$ copies in $S_\curlywedge$, there are at most $k^2$ possibilities to reflect one original edge in $\binom{S}{2}$ as an edge in $\binom{S_\curlywedge}{2}$. This results in $O(k^4)$ branches.

**Lemma 2.** *If $(G = (V, E), \Gamma, S, k)$ an instance of* Split Set Re-Embedding *is a yes-instance with solution $\sigma = (S^*_\curlywedge, \mathit{copies}^*, \mathit{orig}^*, (N_{\dot{v}})_{\dot{v} \in S^*_\curlywedge}, \Gamma^*_{\mathrm{re}})$ and graph $G^*$ corresponding to $\Gamma^*_{\mathrm{re}}$, then there is a branch created by Branching Rule 2 with $E_\curlywedge = E(G^*[S^*_\curlywedge])$.*

*Proof.* By Lemma 1 we know that Branching Rule 1 produces a branch that passes $S^*_\curlywedge$ to Branching Rule 2. Therefore, in this branch the edges in $E_\curlywedge$ must have both endpoints in $S^*_\curlywedge$ in the solution. Consider $E(G^*[S^*_\curlywedge])$ the edge set of the solution $\sigma$ on copies of split vertices. Since Branching Rule 2 attempts all possible edge sets, there exists a branch where $E_\curlywedge = E(G^*[S^*_\curlywedge])$. Thus the branching rule is sound. □

**Obtaining 10$k$-Outerplanarity.** As a second step, we want to reduce the size of our input, or, more precisely, the input drawing $\Gamma$. Note that faces that are not incident to any pistil do not play an important part in the solution as there are no crossing-free edges that can be realized in such a face to achieve our desired adjacencies between copies and pistils. Hence, we use a reduction rule that removes all vertices not adjacent to a petal from $\Gamma$. We then show that this results in a $10k$-outerplanar drawing.

**Reduction Rule 1.** *Let $(G, \Gamma, S, k)$ be a Split Set Re-Embedding (SSRE) instance. Any vertex of $G$ not incident to a petal in $\Gamma$ is removed from $G$ and $\Gamma$, alongside all of its incident edges. This results in the reduced instance $(G', \Gamma', S, k)$.*

**Lemma 3.** *Reduction Rule 1 is sound: Given an instance $I = (G, \Gamma, S, k)$ of SSRE, and applying Reduction Rule 1 to get $I' = (G', \Gamma', S, k)$, it holds that $I$ is a yes-instance if and only if $I'$ is a yes-instance.*

*Proof.* We first show that for a face $f' \in \Gamma'$, for which $f' \notin \Gamma$, there cannot be pistils on its boundary, meaning that any vertex inserted into $f'$ has degree 0 and can be embedded anywhere. Assume there is a pistil $p$ incident to $f'$ and let $F$ be the set of faces incident to $p$ in $\Gamma$. The reduction rule does not remove any vertex incident to a face of $F$ as these faces are all petals. So the faces incident to $p$ in $\Gamma'$ must be the same as those in $\Gamma$. This contradicts that $f'$ is not in $\Gamma$ and hence there cannot be a pistil on $f'$ and a solution for $I'$ is a solution for $I$.

Given a solution $\sigma$ to $I$, we will show that $\sigma$ is also a solution to $I'$. Since every petal of $\Gamma$ is also a face in $\Gamma'$ (we do not remove vertices incident to petals), every embedding in a petal face of $\Gamma$ in $\sigma$ can be replicated in $\Gamma'$. Any copy embedded in a non-petal face of $\Gamma$ must have degree 0 to preserve planarity and can therefore be embedded anywhere. As a result, $I$ is a yes-instance if and only if $I'$ is a yes-instance, and Reduction Rule 1 is sound. □

In the remainder, let $\Gamma_{\text{op}}$ be the drawing obtained from $\Gamma$ after exhaustively applying Reduction Rule 1. We use Lemma 3 every time we apply Reduction Rule 1 to get the following corollary.

**Corollary 2.** Let $I_{\text{p}} = (G, \Gamma, S, k)$ be a Split Set Re-Embedding instance, and let $I_{\text{o}} = (G', \Gamma_{\text{op}}, S, k)$ be the instance obtained from $I_{\text{p}}$ after applying Reduction Rule 1 exhaustively. $I_{\text{p}}$ is a yes-instance if and only if $I_{\text{o}}$ is a yes-instance.

We now introduce the notion of a face path in a drawing $\Gamma$ via a modified bipartite dual graph $D$ of $\Gamma$ whose two sets of vertices are the faces and vertices of $G$ (denoted as *face-vertex* and *vertex-vertex*, respectively) and whose edges link each face-vertex with all its incident vertex-vertices as shown in Fig. 3.5. Paths in $D$ are called *face paths* and they alternate between faces and vertices of $\Gamma$. We define the length of a face path as the number of its vertex-vertices. Using this notion, a drawing is *$k$-outerplanar* if all of its vertex-vertices have a face path to the outer face of length at most $k$. We now show that

applying the reduction rule exhaustively to an instance of Split Set Re-Embedding transforms drawing $\Gamma$ into a $10k$-outerplanar drawing $\Gamma_{\text{op}}$.

**Lemma 4.** *If $(G, \Gamma, S, k)$ is a yes-instance, then $\Gamma_{\text{op}}$ $10k$-outerplanar.*

*Proof.* We show that the shortest face path from any vertex of $\Gamma_{\text{op}}$ to the outer face has length at most $10k$. Let $S^*_{\curlywedge}$ be the set of copies of $S$ embedded in the drawing $\Gamma^*_{\text{re}}$ of a solution. Note that a face of $\Gamma_{\text{op}}$ can be partitioned into multiple faces in $\Gamma^*_{\text{re}}$. We label each face $f$ of $\Gamma_{\text{op}}$ by an arbitrary copy $\dot{v}$ in $S^*_{\curlywedge}$ that is closest to $f$ by length of face paths from the faces partitioning $f$ in $\Gamma^*_{\text{re}}$. The length of a face path in $\Gamma_{\text{op}}$, from $f$ to the face of $\Gamma_{\text{op}}$ that $\dot{v}$ is embedded in, is at most two: The face $f$ is either incident to a pistil ($\dot{v}$ is embedded in $f$, or the face path can traverse the pistil in question to the face in which $\dot{v}$ is embedded), or all its vertices are incident to such a face (the face path traverses one vertex to an incident face with a pistil, and potentially traverses the pistil as well, to reach the face of $\dot{v}$). Additionally, all pistils are covered in $\Gamma^*_{\text{re}}$ and hence in both cases there must be a copy $\dot{v}$ covering the closest pistil.

Since $|S^*_{\curlywedge}| \leq 2k$ there are at most $2k$ distinct labels for the faces of $\Gamma_{\text{op}}$. Let $v$ be a vertex in $\Gamma_{\text{op}}$ and let $P$ be a shortest face path between $v$ and the outer face. We claim that each label appears at most five times in $P$. Assume to the contrary that there are six faces $f_1, f_2, f_3, f_4, f_5, f_6$ with the same label $\dot{u}$ occurring in $P$ in this order. Observe that the distance between $f_1$ and $f_6$ along this (shortest) face path is at least 5. We call $f_{\dot{u}}$ the face in which $\dot{u}$ is embedded in the solution. The faces $f_1$ and $f_6$ must have at most distance 2 from $f_{\dot{u}}$ as otherwise they would be labeled differently. Thus one can find a path from $f_1$ to $f_{\dot{u}}$ of length at most 2, and similarly a path from $f_{\dot{u}}$ to $f_6$ of length at most 2. Hence, there is a shorter path from $f_1$ to $f_6$ which contradicts our claim. Hence, $P$ has length at most $10k$, showing that $\Gamma_{\text{op}}$ is $10k$-outerplanar. □

### 3.4.2 Finding a Sphere-Cut Decomposition

Before we can begin the dynamic programming algorithm, we need to find a sphere-cut decomposition of our drawing. First, we introduce the notion of sphere-cut decomposition,

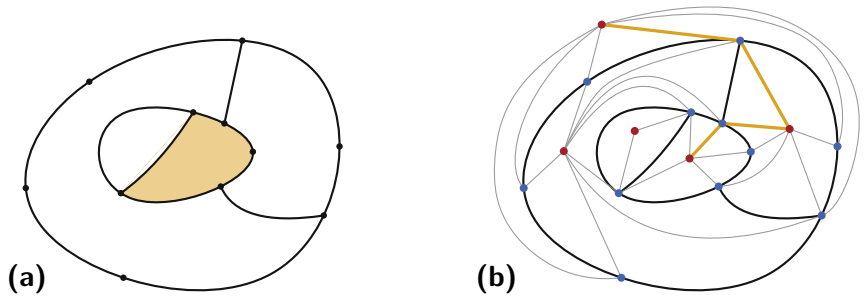



Figure 3.5: **(a)** A graph and **(b)** its modified dual. A shortest face path, drawn with yellow edges in **(b)**, from the yellow face in **(a)** to the outer-face alternates between face-vertices (in red) and vertex-vertices (in blue).

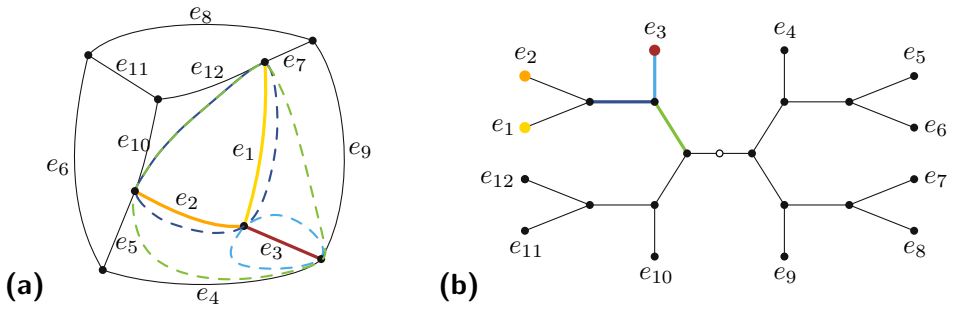



Figure 3.6: **(a)** A graph and **(b)** its sphere-cut decomposition. Each labeled leaf corresponds to the same labeled edge of the graph. The middle set of each colored edge in the tree corresponds to the vertices of the corresponding colored dashed noose in the graph.

and then describe a last transformation applied to our drawing that will ensure we find this decomposition.

A *branch decomposition* of a (multi-)graph $G$ is a pair $(T, \lambda)$ where $T$ is an unrooted binary tree, and $\lambda$ is a bijection between the leaves of $T$ and $E(G)$. Every edge $e \in E(T)$ defines a bipartition of $E(G)$ into $A_e$ and $B_e$ corresponding to the leaves in the two connected components of $T - e$. We define the *middle set* $\mathrm{mid}(e)$ of an edge $e \in E(T)$ to be the set of vertices incident to an edge in both sets $A_e$ and $B_e$. The *width* of a branch decomposition is the size of the biggest middle set in that decomposition. The *branchwidth* of $G$ is the minimum width over all branch decompositions of $G$.

A *sphere-cut decomposition* of a planar (multi-)graph $G$ with a planar embedding $\Gamma$ on a sphere $\Sigma$ is a branch decomposition $(T, \lambda)$ of $G$ such that for each edge $e \in E(T)$ there is a *noose* $\eta(e)$: a closed curve on $\Sigma$ such that its intersection with $\Gamma$ is exactly the vertex set $\mathrm{mid}(e)$ (i.e., the curve does not intersect any edge of $\Gamma$) and such that the curve visits each face of $\Gamma$ at most once (see Figure 3.6). The removal of $e$ from $E(T)$ partitions $T$ into two subtrees $T_1, T_2$ whose leaves correspond respectively to the noose's partition of $\Gamma$ into two embedded subgraphs $G_1, G_2$ (with edges sets respectively $A_e$ and $B_e$). Sphere-cut decompositions are introduced by Seymour and Thomas [ST94], more details can also be found in [MP15, Section 4.6]. The *length* of the noose $\eta(e)$ for an edge $e \in E(T)$ is the number of vertices on the noose (or the size of $\mathrm{mid}(e)$) and it is at most the branchwidth of the decomposition. The drawings in this chapter are defined in the plane, whereas we need drawings on the sphere for sphere-cut decompositions. However, if we treat the outer face of a planar drawing just as any other face, then spherical and planar drawings are homeomorphic.

An $\ell$-outerplanar graph has branchwidth at most $2\ell$ [Bie15]. Moreover, each connected bridgeless planar graph of branchwidth at most $b$ has a sphere-cut decomposition of width at most $b$ and this decomposition can be computed in $O(n^3)$ time where $n$ is the number of vertices (this has been shown by Seymour and Thomas [ST94], see the discussion by Marx and Pilipczuk [MP15, Section 4.6]). We transform any bridge in our newly

obtained graph into a multi edge to ensure that the graph is bridgeless.

**Obtaining a Bridgeless Graph.** While our graph drawing $\Gamma_{\text{op}}$ is already $\ell$-outerplanar (more specifically $10k$-outerplanar), we have to deal with the *bridges*, i.e., edges of $G$ whose removal disconnect $G$. There is no guarantee our graph is bridgeless and even if it was required of the input, Reduction Rule 1 might create bridges. We instead create a new graph $G''$ together with a drawing $\Gamma_{\text{bl}}$, in which for any bridge $(u, v)$, we add a secondary multi edge between $u$ and $v$ and continue to work with the resulting multigraph. Note that adding the edge does not affect the outerplanarity and hence does not affect the bound of the width of the decomposition.

**Lemma 5.** *Given two instances of* Split Set Re-Embedding, *$I_o = (G', \Gamma_{\text{op}}, S, k)$ and $I_b = (G'', \Gamma_{\text{bl}}, S, k)$, which differ only in the graph $G'$ and $G''$ and their respective drawings $\Gamma_{\text{op}}$ and $\Gamma_{\text{bl}}$, where $G''$ is a copy of $G'$ plus a duplicate of every bridge, then $I_o$ is a yes-instance if and only if $I_b$ is a yes-instance.*

*Proof.* Given a solution $\sigma_{\text{o}} = (S^*_\curlywedge, \mathsf{copies}^*, \mathsf{orig}^*, (N_{\dot{v}})_{\dot{v} \in S^*_\curlywedge}, \Gamma^*_{\text{re}})$ to $I_{\text{o}}$, we will show that $\sigma_{\text{o}}$ is also a solution to $I_{\text{b}}$. We can extend $\sigma_{\text{o}}$ to solve $I_{\text{b}}$ as follows. All the faces of $\Gamma_{\text{op}}$ exist in $\Gamma_{\text{bl}}$ and have the same set of vertices incident to them. This means that any split vertex $s$ inserted into a face $f$ of $\Gamma_{\text{op}}$ to cover a set of vertices $N(s)$ can be inserted in the same face of $\Gamma_{\text{bl}}$ and cover the same set of vertices. So $\sigma_{\text{o}}$ is a solution to $I_{\text{b}}$.

Given a solution $\sigma_{\text{b}} = (S^*_\curlywedge, \mathsf{copies}^*, \mathsf{orig}^*, (N_{\dot{v}})_{\dot{v} \in S^*_\curlywedge}, \Gamma^*_{\text{re}})$ to $I_{\text{b}}$, we can extend $\sigma_{\text{b}}$ to solve $I_{\text{o}}$ in the same way as for the previous paragraph with one exception. There may be split vertices that are embedded into a face bounded by two multi-edges $(u, v)$ of $\Gamma_{\text{bl}}$ that does not exist in $\Gamma_{\text{op}}$. Such a split vertex $\dot{s}$ would have one or both of $u, v$ as neighbors. If the split vertex only has one neighbor then it can be embedded in any face incident to that neighbor without disturbing anything. If $N(\dot{s}) = \{u, v\}$, then there is only a single face of $\Gamma_{\text{op}}$ in which they both lie (by virtue of being a bridge). If there are no vertices embedded on that face and we can freely put $\dot{s}$ in it and have it reach its neighborhood without risk of creating a crossing. Otherwise, if there are other copies of split vertices embedded in that face, we can still embed $\dot{s}$ sufficiently close to edge $(u, v)$ and connect it to $u$ and $v$ by two crossing-free edges. This means that after embedding all such vertices, we have found a solution to $I_{\text{o}}$, and with this we showed that $I_{\text{o}}$ is a yes-instance if and only of $I_{\text{b}}$ is a yes-instance. □

As a result, we have a bridgeless graph with a $10k$-outerplanar drawing which thus admits a sphere-cut decomposition of width bounded by $20k$, as discussed before. Moreover, we can compute this decomposition in $O(n^3)$ time [Bie15, MP15, ST94].

### 3.4.3 Initializing the Dynamic Programming

In the previous sections, we used branching and computed a set $S_\curlywedge$ as well as the mappings $\mathsf{copies}$ and $\mathsf{orig}$ for each branch. We now focus on one such branch, and

use dynamic programming on the sphere-cut decomposition $(T_0, \lambda)$ of the bridgeless $10k$-outerplanar graph $G'$, obtained previously, and its drawing $\Gamma'$ to find the remaining elements of the solution: $(N^*_v)_{v \in S_\lambda}$ and $\Gamma^*_{\text{re}}$. We have therefore effectively reduced Split Set Re-Embedding to the following more restricted problem:

**Problem 5** (Split Set Re-Embedding Decomp)**.** Let the following be given: a graph $G = (V, E)$ obtained after Reduction Rule 1 and resolving bridges, a set $S \subseteq V$, an integer $k \geq |S|$, a drawing $\Gamma$ on the sphere of the $10k$-outerplanar bridgeless graph $G' = (V', E') := G[V \setminus S]$, its sphere-cut decomposition $(T_0, \lambda)$, and, as guessed by the initial branching, two mappings $\mathsf{orig}$ and $\mathsf{copies}$ and a graph $G_{S_\lambda}$ on the set of vertices $S_\lambda$.

The task is to decide whether there is a solution $(S^*_\lambda, \mathsf{copies}^*, \mathsf{orig}^*, (N^*_v)_{v \in S^*_\lambda}, \Gamma^*_{\text{re}})$ to the instance $(G, S, \Gamma, k)$ of Split Set Re-Embedding that coincides with the guessed branch, i.e., $S^*_\lambda = S_\lambda$, $\mathsf{orig}^* = \mathsf{orig}$, $\mathsf{copies}^* = \mathsf{copies}$, and $G^*[S^*_\lambda] = G_{S_\lambda}$, where $G^*$ is the corresponding solution graph.

We first transform $T_0$ into a rooted tree $T$ by choosing an arbitrary edge $e_r = (r_1, r_2) \in T_0$ and subdividing it with a new root vertex $r$. This induces parent-child relationships between all the vertices in $T$ (except the new root), and we set for a given vertex $t \in V(T)$ with parent $p$, the noose $\eta(t) = \eta((p, t))$ corresponding to the set $\text{mid}((p, t))$. Since for each $t \in V(T) \setminus \{r\}$ the parent $p$ is unique, we simply use $\text{mid}(t)$ instead of $\text{mid}((p, t))$. Additionally we put $\text{mid}(r_1) = \text{mid}(r_2) = \text{mid}(e_r)$.

The dynamic program works bottom-up in $T$, considering iteratively larger subgraphs of the input graph $G'$. It determines how partial solutions look like on the interface between subgraphs and the rest of $G'$. Saving a single local optimal partial solution, a solution that uses the smallest number of copies, in the corresponding vertex of the decomposition tree is not sufficient. This sub-solution may result in a no-instance when considering the rest of the graph. We therefore keep track of all possible split vertex embeddings that allow us to realize all the missing edges with a crossing-free drawing in the subgraph. Ultimately, we look for two consistent sub-solutions at the root of the decomposition tree to resolve the planar re-embedding of $S$ in $\Gamma$ using at most $2k$ split vertices resulting from at most $k$ splits. We need the following notions to describe the interface between the subgraphs. For a vertex $t \in V(T)$ and the noose $\eta(t)$ associated to it, we define the subgraph $G'_t$ of $G'$ as the subgraph obtained from the union of the edges that correspond to the leaves in the subtree of $T$ rooted at $t$. We say that a subgraph of $G'$ is *inside* the noose $\eta(t)$, if it is a subgraph of $G'_t$. For a noose $\eta(t)$ of $(T, \lambda)$, the *processed* faces are faces of the subgraph $G'_t$ that have all their incident vertices inside or on $\eta(t)$; the *current* faces of $G'_t$ are the faces of $\Gamma$ that have vertices inside, on, and outside of the noose $\eta(t)$, they are not closed faces in $G'_t$.

Nooses in sphere-cut decompositions by definition pass through a face at most once, so it is not possible for a face to have an intersection with a noose that contains more than two vertices. Additionally, since $\Gamma$ is embedded on the sphere, there is no outer face.

Next, we define partial solutions for the subgraphs of $G'$. Intuitively, a partial solution for a subgraph $G'_t$ is a planar drawing $\Gamma'$ of $G'_t$ using a set $S'_\lambda$ of copies that covers all of the pistils inside the noose $\eta(t)$ defining $G'_t$.

**Definition 1.** A *partial solution* for $G'_t$ is a tuple $(S'_\lambda, (N_{\dot{v}})_{\dot{v}\in S'_\lambda}, \Gamma')$, s.t. $S'_\lambda \subseteq S_\lambda$ and:

i for every $\dot{v} \in S'_\lambda$, we have $N_{\dot{v}} \subseteq N_G(\mathsf{orig}(\dot{v}))$,

ii the planar drawing $\Gamma'$ extends $\Gamma$ by embedding each copy $\dot{v} \in S'_\lambda$ in a face or a current face of $G'_t$,

iii for every $v \in \mathsf{orig}(S'_\lambda)$, the family $\{N_{\dot{v}} \mid \dot{v} \in \mathsf{copies}(v)\}$ is a partition of some subset of $N_G(v)$,

iv for each pistil $p$ of $G'$ that is inside (and not on) the noose $\eta(t)$ and for each neighbor $v \in N_G(p) \cap S$ there is a copy $\dot{v} \in \mathsf{copies}(v)$ such that $p \in N_{\dot{v}}$ and $\dot{v} \in N_{\Gamma'}(p)$, and

v the set of edges in $\Gamma'$ with both endpoints in $S'_\lambda$ coincides with the edge set $E_\lambda$ guessed by Branching Rule 2.

The dynamic programming algorithm works bottom-up in $T$. To find a solution at the root, we need to find partial solutions at the leaves and propagate them upwards in the tree. To do so, we need to check if two partial solutions are *compatible*. Notably, we must check that both partial solutions agree on the embedding of the faces of $\Gamma$ that are shared in both nooses (the current faces in the two subgraphs corresponding to the nooses). To achieve this we describe the combinatorial embedding of the split vertices embedded inside that face using a structure called a nesting graph.

Formally, let $\eta$ be a noose, $f$ be a face that is current for $\eta$, and let $S_f \subseteq S'_\lambda$ be a set of copies to be embedded in $f$. A *nesting graph* $C_f$ for $f$ is a combinatorially embedded graph with vertex set $S_f \cup C$, where $C$ is a set of vertices that induces a cycle that encloses all vertices in $S_f$. We call the vertices in $C$ *cycle vertices*. For a graph to be a nesting graph, the following conditions must be satisfied:

i embedded graph $C_f$ is planar and all vertices in $C$ must be incident with its outer face,

ii the vertices of $\cup_{\dot{v}\in S_f} N(\dot{v}) \setminus S_f$ lie in a cycle $C$ and $S_f$ is embedded inside of $C$,

iii each vertex $u \in C$ has exactly one neighbor in $S_f$,

iv for any two vertices $c_1, c_2 \in C$ that have the same neighbor $\dot{s} \in S_f$, it holds that $c_1$ and $c_2$ are not neighbors in $C_f$, and lastly

v $C_f[S_f] = G_{S_\lambda}[S_f]$.

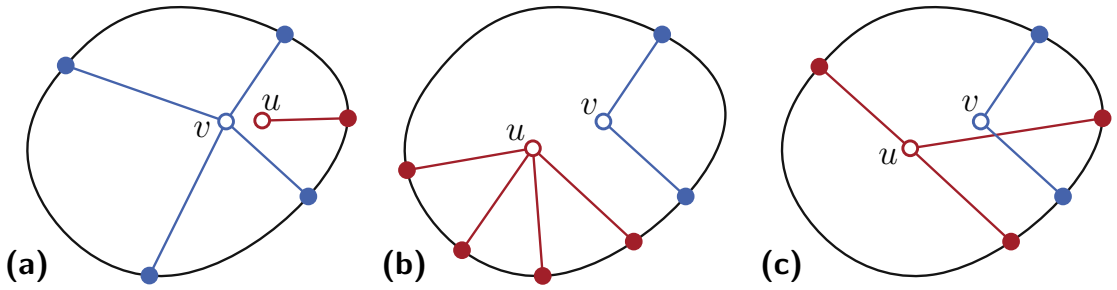



Figure 3.7: **(a)** A degree-one vertex $u$ is compatible with any other vertex, here only $v$. **(b)** Vertex $u$ is still compatible with $v$. **(c)** Vertices $u$ and $v$ are not compatible.

The cycle is a combinatorial representation of the boundary of a face and its vertices represent a subset of the vertices of the boundary of $f$ (possibly merged). Intuitively, the nesting graph describe how to embed copies inside a face in a solution. To draw the edges between the pistils in the current face $f$ and the copies in $S_f$, we can imagine $C_f$ embedded inside of $f$ and attempt to draw crossing-free edges from the vertices of $C$ to the pistils of $f$.

To describe the combinatorial embeddings of the vertices inside of the cycle of nesting graphs, we introduce the notion of *compatibility* (see Figure 3.7).

**Definition 2.** Given a face $f$ and two copies $\dot{u}, \dot{v} \in S_\lambda$ to be embedded in $f$ with their respective neighborhoods $N(\dot{u})$ and $N(\dot{v})$ incident to $f$, we say that $\dot{v}$ *is compatible with* $\dot{u}$ in $f$ if in the cyclic ordering of $N(\dot{u})$ and $N(\dot{v})$ around $f$, the respective neighborhoods not alternate, meaning it is possible in a traversal of $f$ to encounter first vertices in $N(\dot{u})$ and no vertices in $N(\dot{v})$, and then vertices in $N(\dot{v})$ and not in $N(\dot{u})$.

To propagate the partial solutions of two child vertices to the parent, we must carefully consider the nesting graphs of the current faces shared by both children. Since current faces of $G'_t$ are not closed faces in $\eta(t)$, we want to specify which part of a nesting graph $C_f$ is used in a current face $f$ to cover incident pistils. To achieve this we keep track of only the first and last vertices $p_\mathrm{s}, p_\mathrm{e}$ on the cycle $C$ of $C_f$ that connect to pistils in $f$. That is, a clockwise traversal of $C$ from $p_\mathrm{s}$ to $p_\mathrm{e}$ visits all vertices used to cover pistils of $G'_t$. If only a single pistil is covered, then $p_\mathrm{s} = p_\mathrm{e}$; if no pistil is covered $p_\mathrm{s}$ and $p_\mathrm{e}$ can be undefined.

While all pistils inside the noose $\eta(t)$ of $G'_t$ are covered in a partial solution, the vertices on $\eta(t)$ can have missing neighbors. That is, for a pistil $p$ on $\eta(t)$ with neighborhood $N_G(p)$ in $G$ and $N_{\Gamma'}(p)$ in $\Gamma'$ (the drawing of the partial solution), and $v \in S \cap N_G(p)$ such that $\mathsf{copies}(v) \cap N_{\Gamma'}(p) = \emptyset$, vertex $v$ is a *missing neighbor* of $p$. We define $X_t(p)$ as the set of missing neighbors of $p$.

Using the elements described above, we build a *signature* for a node $t \in T$ modeling a possible solution.

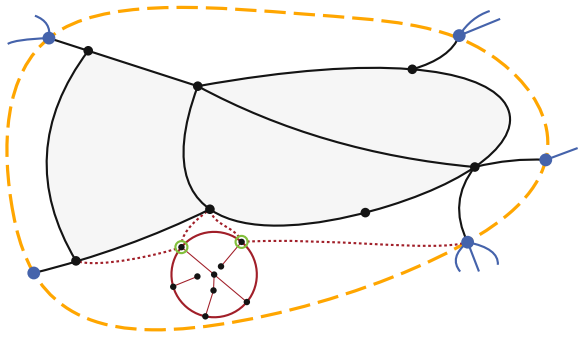


Figure 3.8: The information stored in a signature of the partial solution inside the orange dashed noose: copies in $S_{\text{in}}$ are used in the grey faces, blue noose vertices who have missing neighbors (outgoing edges outside the noose) are stored in $N_\eta$, in red an example of a nesting graph for a face traversed by the noose, with four dotted edges connecting to the cycle and the vertices described by $M$ in green.

**Definition 3.** A *signature* on $t \in V(T)$ is a tuple $(S_{\text{in}}(t), C_{\text{out}}(t), M_t, N_\eta(t))$ where:

1. $S_{\text{in}}(t) \subseteq S_\curlywedge$,
2. $C_{\text{out}}(t)$ is a set containing a nesting graph $C_f$ for each current face $f$ of $G'_t$ such that no two of the graphs share a vertex in $S_\curlywedge$,
3. $M_t \colon C_{\text{out}}(t) \to V(C_{\text{out}}(t)) \times V(C_{\text{out}}(t))$ maps each nesting graph to a pair $(p_{\text{s}}, p_{\text{e}})$, and
4. $N_\eta(t)$ is a list of $|\operatorname{mid}(t)|$ sets of vertices, denoted by $\langle X_t(p)\rangle_{p\in\operatorname{mid}(t)}$.

Intuitively, for a given signature on $t \in V(T)$, the set $S_{\text{in}}(t) \subseteq S_\curlywedge$ tracks the split vertices embedded in the processed faces of $G'_t$, graph $C_{\text{out}}(t)$ and $M_t$ track the embedding information in current faces, and $N_\eta(t)$ tracks the missing neighbors of the noose vertices (see Figure 3.8).

We now show that every partial solution has a corresponding signature. For a partial solution $(S'_\curlywedge, (N_{\dot v})_{\dot v\in S'_\curlywedge}, \Gamma')$ for $G'_t$ we construct a signature tuple $(S_{\text{in}}(t), C_{\text{out}}(t), M_t, N_\eta(t))$ (or $(S_{\text{in}}, C_{\text{out}}, M, N_\eta)$ when $t \in V(T)$ is clear from context) as follows. The set $S_{\text{in}}$ is composed of all vertices of $S'_\curlywedge$ embedded in processed faces of $G'_t$. To construct $N_\eta$, we create a set $X_t(p) \in N_\eta$ for each noose vertex $p$, and add to $X_t(p)$ all neighbors in $S$ for which no copy is adjacent to $p$ in $\Gamma'$: $X_t(p) = (N_G(p) \cap S) \setminus \mathsf{orig}(N_{\Gamma'}(p) \cap S'_\curlywedge)$. Lastly, for a current face $f$ of $G'_t$ where $S_f \subseteq S'_\curlywedge$ are the copies embedded in $f$, we find the nesting graph $C_f$ by transforming $f$ and the graph induced by $S_f$: If $|S_f| = 1$, graph $C_f$ will consist of that vertex embedded inside a 2-cycle, with outgoing edges from the split vertex to both cycle vertices. If $|S_f| \geq 2$, the construction is as follows (see Figure 3.9).

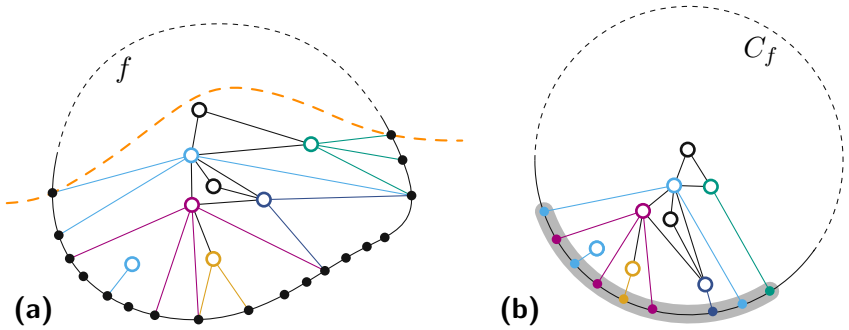



Figure 3.9: **(a)** A face $f$ and copies inside the orange noose, and **(b)** the corresponding nesting graph $C_f$ with the interval described by $M$ highlighted in grey. The two light blue vertices represent two different copies of the same removed vertex. Copies in $C_f$ have no edges to copies in other nesting graphs.

- For each $v \in V(G'_t)$ such that $N(v) \cap S'_\lambda = \emptyset$ we contract one of its outgoing edges in $f$. We repeat this process until all remaining vertices of $f$ are incident with a vertex of $S'_\lambda$.
- For each $u \in V(G'_t)$ we let $\dot{s}_1, \ldots, \dot{s}_i$ be the ordering of $N(u) \cap S'_\lambda$ clockwise around $u$. We replace $u$ by the path $(u_1, u_2), \ldots, (u_{i-1}, u_i)$ on which we then add the edges $(\dot{s}_1, u_1), \ldots, (\dot{s}_i, u_i)$. Thus each vertex on $f$ is adjacent to at most one vertex of $S'_\lambda$.
- For $e = (u_1, u_2)$ incident to $f$, if there is a split vertex $\dot{s} \in S'_\lambda$ such that $(u_1, \dot{s}), (u_2, \dot{s})$ in $\Gamma'$, we contract $e$ such that the new vertex obtained has a single edge to $\dot{s}$.

Since $G'_t$ is bridgeless and we initialized $C_f$ as $f$, $C_f \setminus S_f$ is necessarily a cycle $C$. To obtain $M$, for each graph in $C_f$ we find the section of $C$ used for the coverage of $G'_t$. We determine the first and last vertex in clockwise ordering on cycle $C$, here $p_s$ and $p_e$, and set $M_t(C_f) = (p_s, p_e)$. The obtained signature is the *signature of the partial solution*. We say that each signature for which a partial solution exists is *valid*.

During dynamic programming we enumerate all signatures in linear time, and hence we first determine how many distinct signatures can exist. To bound the number of different signatures, we need an upper bound on the number of vertices in a nesting graph.

**Lemma 6.** *A nesting graph with $s$ split vertices embedded inside the cycle $C$ has at most $2s$ vertices on $C$.*

*Proof.* Let $C_f = (V, E)$ be a nesting graph, where $V = C \cup S_f$ with $C$ being the cycle defining the outer face and $S_f$ the vertex set embedded inside the cycle and let $|S_f| = s$. We remove all edges $e = (u, v) \in E$ such that $u, v \in S_f$ and all vertices in $S_f$ that have degree zero after the removal. We obtain a new set of faces $F$. Euler's formula tells us that for this modified nesting graph $G' = (V', E')$ we have $|V'| - |E'| + |F| = 2$. We first

notice $|V'| \leq s + |C|$ and thus

$$s + |C| - |E'| + |F| \geq 2. \tag{3.1}$$

Considering that $E' = \{(u, v), u \in C, v \in C\} \cup \{(u, v), u \in C, v \in S_f\}$, that by property (i) of nesting graphs $|\{(u, v), u \in C, v \in C\}| = |C|$, and that $|\{(u, v), u \in C, v \in S_f\}| = |C|$ (from (ii)), we obtain $|E'| = 2|C|$. Plugging this into Eq. (3.1), we have

$$\begin{aligned} s + |F| &\geq 2 + |C| \\ s + |F| - 2 &\geq |C|. \end{aligned}$$

We lastly show that $|F| \leq \frac{|C|}{2}$. Assume for a contradiction that a face $f \in F$ that is not the outer face has only one edge $e = (u, v)$ of $C$ on its boundary. We traverse $f$ starting from $v$, visiting its neighbor $v_s$ different from $u$. Note that $v_s \in S_f$. Then as we have removed all edges incident to $v_s$ having their other endpoint in $S_f$, the next vertex $v_c$ is necessarily on the cycle $C$. Considering property (iv) of nesting graphs we find that $v_c \neq u$. Thus our traversal continues and property (iii) implies that the next vertex must be a vertex on the cycle different from $u$ and $v$, which contradicts our assumption that an inner face of $C_f$ only has a single cycle edge incident to itself.

Hence, $|C| \leq s - 2 + \frac{|C|}{2}$, giving us $|C| \leq 2s - 4 \leq 2s$. □

Next we bound the number of distinct signatures for $k$ split operations.

**Lemma 7.** *The number $N_s(k)$ of valid signatures is upper bounded by $2^{O(k^2)}$. Moreover, we can enumerate a superset of signatures that includes all valid signatures in $\mathcal{O}(N_s(k))$ time.*

*Proof.* Consider a valid signature. We first count the number of all possible sets $S_{\text{in}}$ included in the signature. These are necessarily subsets of $S_\curlywedge$. Thus, there are almost $2^{2k}$ such sets.

To find the number of the possible sets $N_\eta$, we compute the set of all candidate vertices that are neighbors to the noose vertices. As our input graph must be a $10k$-outerplanar graph and we know that those have sphere-cut decompositions of branchwidth upper bounded by $20k$ [Bie15, Lemma 3], there are at most $20k$ vertices on a noose in the decomposition. Since a single vertex has at most $k$ split neighbors, the set is of size at most $20k^2$. This means that there are at most $2^{20k^2}$ sets $N_\eta$.

To find the number of all the possible sets $C_{\text{out}}$ of nesting graphs, consider constructing an auxiliary graph $\hat{C}$ that consists of the union of all nesting graphs in $C_{\text{out}}$ connected by edges in an arbitrary tree-like fashion. To find an upper bound for the number of sets $C_{\text{out}}$ it is enough to find an upper bound on the number of (combinatorially) embedded graphs $\hat{C}$. By Lemma 6 and since no two nesting graphs in $C_{\text{out}}$ share a vertex in $S_\curlywedge$ (by the definition of a signature), it follows that $|V(\hat{C})| \leq 4k$ (recall that $|S_\curlywedge| \leq 2k$).

Consider taking $\hat{C}$ and triangulating it; in this way it becomes triconnected. Thus $\hat{C}$ is a partial drawing of a triconnected embedded planar graph with at most $4k$ vertices, that is, $\hat{C}$ can be obtained from this graph by omitting vertices and edges. The number of partial drawings of a fixed triconnected embedded planar graph with at most $4k$ vertices is at most $2^{O(k)}$ because such a graph contains $O(k)$ edges (as it is not the full drawing). It remains to find an upper bound on the number of triconnected combinatorially embedded planar graphs. Since each triconnected planar graph with $4k$ vertices has $O(k)$ different combinatorial embeddings (the embedding being fixed up to the choice of the outer face), an upper bound is $O(k)$ times the number of planar graphs with $4k$ vertices. To bound the number of planar graphs with $4k$ vertices, recall that each planar graph is 5-degenerate, that is, it admits a vertex ordering such that each vertex $v$ has at most 5 neighbors before $v$ in the ordering. Thus, there are at most $((4k)^5)^{4k}$ planar graphs with $4k$ vertices because all planar graphs on a given vertex set can be constructed by, for each vertex $v$, trying all possibilities to select the at most five neighbors of $v$ occurring before $v$ in the degeneracy ordering. Overall, we obtain an upper bound of $2^{O(k)} \cdot O(k) \cdot O(k)^{O(k)} = 2^{O(k \log k)}$ for the number of sets $C_{\text{out}}$. As many of the graphs created are not nesting graphs, if a graph is missing any of the properties of the nesting graph, it is discarded.

We lastly need to find all mappings $M_t$ for the nesting graph set. The trees have at most $4k$ vertices, which means in the worst case a cycle has $8k$ vertices. This gives us at most $\binom{8k}{2}$ vertex pairs per nesting graph (as well as $\emptyset$) as possible outputs for every nesting graphs.

This means that there are at most $N_s(k) = 2^{2k} \cdot 2^{20k^2} \cdot 2^{O(k \log k)} \cdot \binom{8k}{2} = 2^{O(k^2)}$ valid signatures for a given noose. Moreover, the upper bound derived above is constructive and yields a linear-time algorithm for enumerating signatures, including all valid signatures. □

### 3.4.4 Dynamic Programming

We first give an overview of our dynamic programming approach, starting with how the valid signatures at the root are found. In each branch obtained after Branching Rule 1 and 2, we perform bottom-up dynamic programming on the tree $T$ of the sphere-cut decomposition. We want to find a valid signature at the root node of $T$, which corresponds to a solution for the whole input graph $G'$, and we start from the leaves of $T$. Each leaf corresponds to an edge $(u_1, u_2)$ of the input graph $G'$, for which we consider all enumerated signatures and check if a signature is valid and thus corresponds to a partial solution. Such a partial solution should cover all missing neighbors of $u_1$ and $u_2$ that are not in $N_\eta = \{X(u_1), X(u_2)\}$, using for each incident face $f$ the subgraph of $C_f \in C_{\text{out}}$ as specified by $M$.

For internal nodes of $T$ we merge some pairs of valid child signatures corresponding to two nooses $\eta_1$ and $\eta_2$. We merge if the partial solutions corresponding to the child signatures can together form a partial solution for the union of the graphs inside $\eta_1$ and

$\eta_2$. The signature of this merged partial solution is hence valid for the internal node when

1. faces not shared between the nooses do not have copies in common,
2. shared faces use identical nesting graphs and
3. use disjoint subgraphs of those nesting graphs to cover pistils, and
4. noose vertices have exactly a prescribed set of missing neighbors.

As we will see, these conditions suffice and thus we can find valid signatures for all nodes of $T$ and notably for its root. For each node $t \in T$ the valid signatures are stored in a corresponding table $D(t)$. If we find a valid signature for the root, a partial solution $(S'_\lambda, (N_v)_{v \in S'_\lambda}, \Gamma')$ must exist. In $\Gamma'$ all pistils are covered and it is planar, as the nesting graphs are planar and they represent a combinatorial embedding of copies that together cover all pistils. The running time for every node of $T$ is polynomial in the number of signatures $N_s(k)$, thus, over all created branches Split Set Re-Embedding is solved in $2^{O(k^2)} \cdot n^{O(1)}$ time.

The next three sections will each explain part of our dynamic-programming approach, starting with the leaf nodes of the sphere-cut decomposition tree $T$ and finishing with the proof for Theorem 5.

**Finding Valid Signatures for Leaf Nodes.** By definition, the leaf nodes of $T$ form a bijection with the edges of the drawing $\Gamma$. The first step of the dynamic programming algorithm is to find the set of valid signatures on the leaf nodes of $T$. To do this, we loop over each possible signature for each leaf $t \in V(T)$, and check whether it is valid. In the following, we show how we determine whether a signature for a leaf $t \in V(T)$ is a valid signature for $t$, that is, the signature corresponds to a partial solution for the subinstance induced by the edge of $\Gamma$ that corresponds to $t$. The signatures we created by enumerating all possibilities as described in Lemma 7 are not all useful, so we check beforehand that they satisfy the necessary properties. From the non-discarded leaf signatures we can progress our dynamic programming algorithm to the inner nodes of $T$.

For a leaf node $t \in T$ with corresponding edge $e = (u, v)$ in $\Gamma$ and given a signature $t$ $\mathrm{sig} = (S_{\mathrm{in}}(t), C_{\mathrm{out}}(t) = \{C_1, C_2\}, M_t, N_\eta(t) = \{X_t(u), X_t(v)\})$, the algorithm will proceed in two steps to cover $u$ and $v$ by embedding vertices in the two faces of $\Gamma$ incident to $e$. It will first check elements of the signature and whether there are pistils on the noose to assess trivially valid and invalid signatures. Then it will execute a routine based on branching and traversing the cycle of the nesting graph to assess the remaining signatures. We first define the set $N_u(t) = N_G(u) \setminus X_t(u)$ to contain the neighbors of $u$ that should cover $u$, as they are not in the missing neighbors set in $G'_t$ and we analogously define $N_v(t)$. The trivial checks are the following:

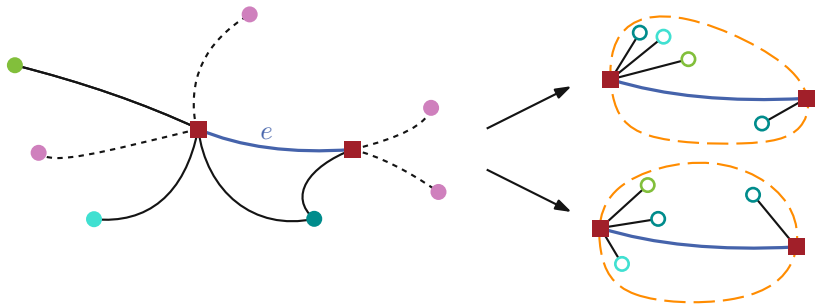



Figure 3.10: The edge $e$ corresponding to the current leaf node $t \in T$ in blue, and its endpoints (both pistils) as red squares. The filled colored disk correspond to its neighborhood in $S$. The vertices of $N_\eta(t)$ are drawn in pink and are ignored by $\mathcal{A}$. The remaining vertices can be covered using any cyclic ordering and can be embedded in any of the two faces incident to $e$. On the right two examples of a possible coverage with copies, shown as circles, that $\mathcal{A}$ looks for in their noose (orange).

1. If $S_{\text{in}} \neq \emptyset$ then the signature is not valid – as $G'_t$ is only an edge, there is no processed face.
2. If $N_u(t) = N_v(t) = \emptyset$, then the signature is valid and we put sig $\in D(t)$: There is no vertex to cover in the subgraph, hence the signature is valid (even if $M_t \neq \emptyset$ the vertices in the nesting graph will just not be used).
3. If $N_u(t), N_v(t) \neq \emptyset$, but $C_{\text{out}} = \{\emptyset, \emptyset\}$ or $M_t = \emptyset$, then there is no available vertex to cover the pistils, hence the signature is not valid.

For the general case, we now introduce algorithm $\mathcal{A}$ to check the validity of the signatures on leaf nodes, using the following notation. Let $p^1_{\text{s}}, p^1_{\text{e}} \in C_1$ and $p^2_{\text{s}}, p^2_{\text{e}} \in C_2$ such that $M_t(C_1) = (p^1_{\text{s}}, p^1_{\text{e}})$ and $M_t(C_2) = (p^2_{\text{s}}, p^2_{\text{e}})$. Let $f_1$ be the face incident to $e$ in $\Gamma$ associated with $C_1$. A clockwise traversal $f_1$ encounters the endpoints of $e$ one after the other in a specific order: w.l.o.g. we assume here that $u$ is visited right before $v$, meaning for the other face $f_2$ incident to $e$, a clockwise traversal finds $v$ right before $u$.

In its execution, algorithm $\mathcal{A}$ first branches over all possible partitions of $N_u(t) \cup N_v(t)$ into two sets $S_1, S_2$. Then, for each branch, it creates two new branches, assigning (1) $S_1$ to $C_1$ and $S_2$ to $C_2$ and (2) vice versa as shown in Fig. 3.10.

Next, algorithm $\mathcal{A}$ creates a branch for each ordering of $N_u(t) \cap S_1$. In each branch, the algorithm proceeds as follows. It creates a queue $Q_1$ from the ordering of $N_u(t) \cap S_1$ and a queue $Q_2$ containing the elements of the cycle of $C_1$ in the order of a clockwise traversal starting with $p^1_{\text{s}}$. It then checks whether the first element $q_1$ of $Q_1$ is equal to the first element $q_2$ of $Q_2$. If so, $q_1$ is removed from $Q_1$. Intuitively, this corresponds to covering the pistil $u$ by a copy of $q_1$ connected to cycle vertex $q_2$. Otherwise, the cycle vertex $q_2$ is not connected to a copy of the current missing neighbor $q_1$, thus $q_2$ is removed from $Q_2$. It repeats the checks and removals of the first elements until either (1)

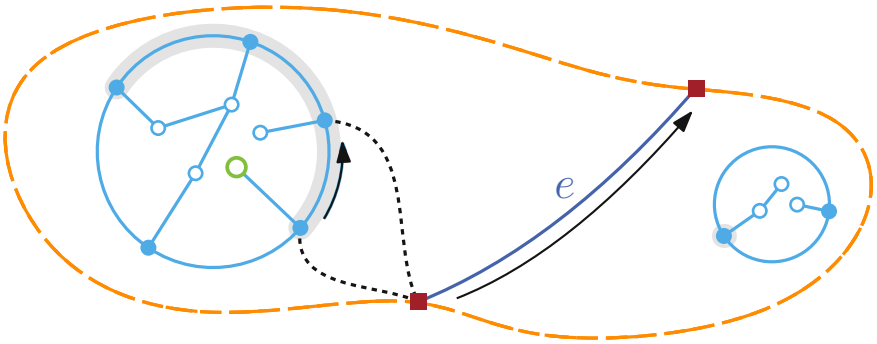



Figure 3.11: Algorithm $\mathcal{A}$ checks if the cyan nesting graphs correspond to a valid signature for the blue edge in the orange noose. The path between $p_s^1$ and $p_e^1$ is highlighted in gray. First, it checks if the green copy is the desired first neighbor of the red pistil. Then it continues down the gray path, or moves on to the second red pistil if all desired vertices have been found. When the desired coverage has been found for the first face without exhausting all vertices on the gray path, the process is repeated in the second face, using the other (cyan) nesting graph.

$Q_1$ becomes empty or (2) $Q_2$ becomes empty. In the second case, the algorithms discards the current branch, since not all neighbors of pistil $u$ are covered. In the first case, the algorithms proceeds in the same way with $N_v(t) \cap S_1$ and $C_2$. If in at least one branch for an ordering of $N_u(t) \cap S_1$, the algorithm never reaches case (2), the algorithm accepts and puts sig $\in D(t)$. However, if all branches are discarded then the signature is not valid. Thus, we fill $D(t)$ with all signatures found to be valid.

**Lemma 8.** *For each leaf node $t \in T$, algorithm $\mathcal{A}$ correctly computes the table of valid signatures $D(t)$.*

*Proof.* For a leaf $t \in T$ such that $\lambda(t) = e$ where $e = (u, v)$ in $E(G')$, we show that for sig $= (S_{\text{in}}(t), C_{\text{out}}(t), M_t, N_\eta(t))$ algorithm $\mathcal{A}$ finds sig $\in D(t)$ if and only if the input signature is valid, meaning we can find a partial solution $\sigma = (S'_\lambda, (N_{\dot{v}})_{\dot{v} \in S'_\lambda}, \Gamma')$ from sig. In the following proof, for ease of reading, we identify the vertices on the cycle of a nesting graph with corresponding split neighbor in that graph.

Given a partial solution $\sigma$ for the signature sig, and embedded subgraph $G'_t = (\{u, v\}, e)$, we show that algorithm $\mathcal{A}$ sets sig $\in D(t)$. Let $f_1$ and $f_2$ be the faces incident to $e$ in $\Gamma$, and let $C_1$ and $C_2$, respectively, be their assigned nesting graphs. By definition of the signature of a partial solution, since $G'_t$ has no processed faces, necessarily $S_{\text{in}}(t) = \emptyset$. Additionally, if $N_u(t) = N_v(t) = \emptyset$, necessarily $C_{\text{out}}(t) = \{\emptyset, \emptyset\}$. Similarly if $N_u(t), N_v(t) \neq \emptyset$, then $C_{\text{out}}(t), M_t \neq \emptyset$. Lastly all edges between split vertices in the solution are fully in $f_1$ or $f_2$, therefore the preliminary checks do not discard the signature. If $N_u(t), N_v(t) \neq \emptyset$ in the partial solution, $u$ and $v$ have a subset of their neighborhood in $f_1$ and/or $f_2$. Algorithm $\mathcal{A}$ attempts all partitions of $N_u(t) \cup N_v(t)$ into two sets, hence two branches will necessarily correspond to the partition of the solution. Additionally, since both

options of assigning a set of the partition to a face are considered, one branch will assign the correct subsets of $S'_\lambda$ to the correct faces. We place ourselves in the search tree nodes corresponding to this path. W.l.o.g., by definition of the signature of a partial solution, the clockwise traversal of the neighborhood of $u$ and $v$ in $f_1$ corresponds to a sub-sequence of a clockwise traversal of the vertices on the cycle of $C_1$ between $p^1_s$ and $p^1_e$. This sub-sequence of cycle $C_1$ will correspond to one of the orders we branch over: since a solution exists, this order is one of the possible orders of $N_u(t) \cap S_1$, $N_v(t) \cap S_1$, hence, each search will always find a copy of the neighbor it is trying to cover the pistil with, meaning $\mathcal{A}$ finds the signature of a partial solution to be valid.

Given a signature sig such that $\mathcal{A}$ finds sig $\in D(t)$, we show that sig corresponds to a partial solution $\sigma = (S'_\lambda, (N_{\dot{v}})_{\dot{v} \in S'_\lambda}, \Gamma')$. We set $V(C_1) \cup V(C_2) = S'_\lambda$. If $N_u(t) = N_v(t) = \emptyset$, sig $\in D(t)$: If no endpoints of the edge have to be covered, $S'_\lambda = \emptyset$, there are no neighborhoods to cover, and the drawing of the input edge is necessarily planar. Hence, if $N_u(t) = N_v(t) = \emptyset$, then $\mathcal{A}$ correct puts sig $\in D(t)$. Otherwise, $\mathcal{A}$ returns sig $\in D(t)$ when a branch of the search tree encounters all the neighbors it searches for before completing the nesting graph cycle traversal in both faces. In this case, we construct the partial solution in the following way. We place the nesting graphs in their corresponding face. We follow the path of the search tree that was able to cover each pistil(s). When encountering a search tree node, we draw an edge between the cycle vertex found by the traversal of the nesting graph and the pistil it is in the neighborhood of. Because both traversals (the edge in the noose and the cycle) are in the same direction, the drawing is planar. Indeed there could only be a crossing between an edge that has $u$ as an endpoint and an edge that has $v$ as an endpoint, but since their neighborhoods are successive in the traversal of the nesting graph (they do not alternate), this is impossible. We can then remove the cycle edges and contract the edges between the pistils and the cycle vertices. The nesting graph is planar and these operations preserve planarity ensuring the final drawing is planar. The set of vertices $S_f$ inside the cycle of both nesting graphs $C_1, C_2$ is such that for every $s \in S$ that has a copy in $S_f$, we have that $\{N_{\Gamma'}(\dot{s}) \mid \dot{s} \in \mathsf{copies}(s) \cap S_f\}$ is a partition of some subset of $N_G(s)$, more specifically, a subset of $\{u, v\}$. There are no pistils inside of the noose, $S'_\lambda \subseteq S_\lambda$ and for every $\dot{s} \in S'_\lambda$ we have $N_{\dot{s}} \subseteq N_G(\mathsf{orig}(\dot{s}))$, showing that sig is a partial solution. Thus, $\mathcal{A}$ correctly puts sig $\in D(t)$. □

We now derive the running time.

**Lemma 9.** *For a leaf node $t \in V(T)$, we can fill $D(t)$ in $\mathcal{O}(N_s(k)k2^{2k}2k!)$ time.*

*Proof.* The trivial checks can be done in constant time. For the main part of the algorithm, there are at most $k$ missing neighbors to $u$ and $v$ each, meaning $2^{2k-1}$ partitions of the missing neighbors, giving $2^{2k}$ branches as each set creates one branch for each face assignment. Each of these branches will further branch into all the possible orders of their vertex set, meaning a single branch will further create at most $2k!$ branches. Each branch traverses the two sub-cycles once. Since any cycle has at most $4k$ vertices this

takes $\mathcal{O}(k)$ time. Overall for a given signature it takes $\mathcal{O}(k2^{2k}2k!)$ time to check whether it is valid, and thus it takes $\mathcal{O}(N_s(k)k2^{2k}2k!)$ time to fill the table for a leaf node. □

This concludes the description of the algorithm for computing all valid signatures at the leaf nodes of the decomposition tree $T$.

**Finding Valid Signatures for Internal Nodes.** In this section, we describe an algorithm that fills the table of valid signatures for internal nodes of sphere-cut decomposition $T$. For $t, c_1, c_2 \in V(T)$ such that $c_1, c_2$ are the children of $t$ and given tables $D(c_1)$ and $D(c_2)$ of valid signatures for $c_1, c_2$ respectively, we will describe an algorithm that checks for each valid signature pair whether there exists a corresponding valid signature for $t$.

Given the signatures $s_1, s_2$ of two children $c_1, c_2$ of the same node $t$, we introduce algorithm $\mathcal{B}$ that tests the two signatures for the following four properties. If the pair $s_1, s_2$ has all the required properties, $\mathcal{B}$ finds the signatures to be *consistent* and combines them to compute a signature $s_t = \mathcal{B}(s_1, s_2)$ for the parent node. For $i \in [2]$ we let $s_i = (S_{\text{in}}(c_i), C_{\text{out}}(c_i), M_i, N_\eta(c_i))$. The tests are the following.

1. Firstly, we require $S_{\text{in}}(c_1) \cap S_{\text{in}}(c_2) = \emptyset$, $S_{\text{in}}(c_1) \cap S_{\text{out}}(c_2) = \emptyset$, and $S_{\text{out}}(c_1) \cap S_{\text{in}}(c_2) = \emptyset$, where $S_{\text{out}}$ is the union of all the split vertices of the graphs of $C_{\text{out}}$. This ensures vertices shared by both signatures can only belong to nesting graphs (see Fig. 3.12(a)).

2. Next, consider the set of vertices $I = (\text{mid}(c_1) \cup \text{mid}(c_2)) \setminus \text{mid}(t)$ that are on the noose of both children $c_1, c_2$ and not on the noose of $t$. We require for every $v \in I$ with missing neighbors sets $X_{c_1}(v) \in N_\eta(c_1)$ and $X_{c_2}(v) \in N_\eta(c_2)$ that $X_{c_1}(v) \cap X_{c_2}(v) = \emptyset$ (see Fig. 3.12(b)).

3. We then check that $S_{\text{out}}(c_1) \cap S_{\text{out}}(c_2)$ contains only vertices in nesting graphs corresponding to faces that intersect both nooses. That is for all faces $f \in \eta(t)$ current in $\eta(c_1)$ and $\eta(c_2)$ with $C_f(c_1) \in C_{\text{out}}(c_1), C_f(c_2) \in C_{\text{out}}(c_2)$ the corresponding nesting graphs should be equivalent, and hence $C_f(c_1) = C_f(c_2)$ (see Fig. 3.12(c)).

4. Lastly, for all faces $f$ current in both $s_1$ and $s_2$, given the mappings $M_1$ of $s_1$ and $M_2$ of $s_2$ such that $M_1(C_f) = (p_{\text{s}}^1, p_{\text{e}}^1)$ and $M_2(C_f) = (p_{\text{s}}^2, p_{\text{e}}^2)$, we check that a clockwise traversal of the cycle of $C_f$ from $p_{\text{s}}^1$ encounters first $p_{\text{e}}^1$ then $p_{\text{s}}^2$ and finally $p_{\text{e}}^2$ (see Fig. 3.12(d)), the order is not strict, $p_{\text{s}}^1 = p_{\text{e}}^1 = p_{\text{s}}^2 = p_{\text{e}}^2$ or any equality of consecutive elements is allowed.

Any signature pair $s_1, s_2$ that fails any of the above tests is not consistent and no parent signature is computed. Otherwise, to compute the combination, Algorithm $\mathcal{B}$ does the following. The set $C_{\text{out}}(t)$ is set to $C_{\text{out}}(c_1) \cup C_{\text{out}}(c_2) \setminus C$ where $C$ corresponds to the

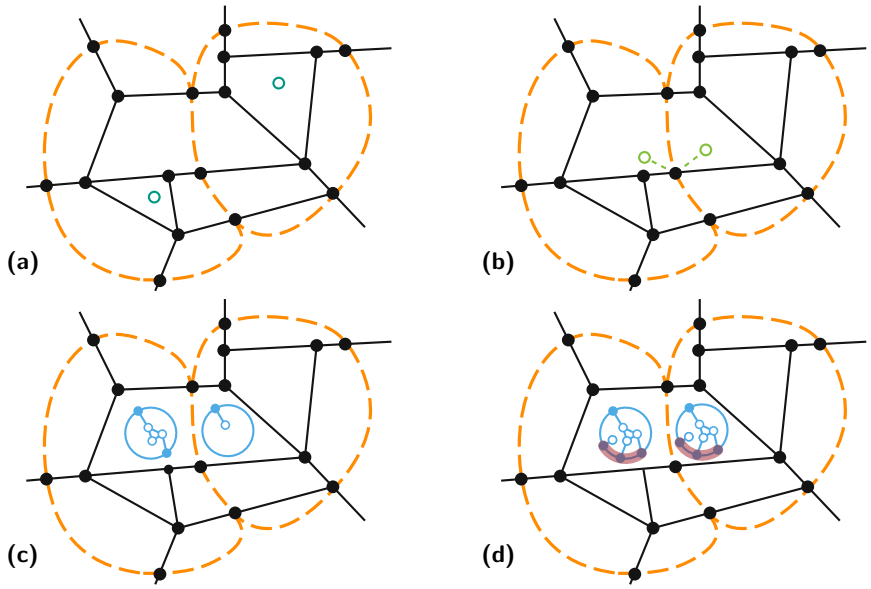



Figure 3.12: Algorithm $\mathcal{B}$ checks for consistent signatures and rejects a pair for the orange noose if **(a)** the corresponding partial solutions both use the same green copy, **(b)** a vertex is a missing neighbor in both partial solutions (in light green), **(c)** a face uses two different nesting graphs, and **(d)** while the nesting graphs are the same, the used cycle vertices are interleaved. A combined signature for such a pair of signatures is invalid.

nesting graphs of faces current in $\eta(c_1)$ and $\eta(c_2)$ and processed in $\eta(t)$. Let $S_f$ be the set of vertices in $C$ and set $S_{\text{in}}(t) = S_f \cup S_{\text{in}}(c_1) \cup S_{\text{in}}(c_2)$. The new mapping $M_t$ is a combination of $M_1$ and $M_2$ on the nesting graphs for the faces that are current in $\eta(t)$. For such a current face $f$ there are four candidate vertices: $p^1_{\text{s}}, p^1_{\text{e}}, p^2_{\text{s}}, p^2_{\text{e}}$. The two vertices that are not on the noose $\eta(t)$ are removed. If $p^1_{\text{s}}, p^2_{\text{e}}$ remain, $M_t(C_f) = (p^1_{\text{s}}, p^2_{\text{e}})$, otherwise, $p^1_{\text{e}}, p^2_{\text{s}}$ remain and $M_t(C_f) = (p^2_{\text{s}}, p^1_{\text{e}})$. Lastly, $N_\eta(t)$ is composed from $N_\eta(c_1)$ and $N_\eta(c_2)$ for all vertices in $\text{mid}(t)$. For $p \in \text{mid}(c_1) \cap \text{mid}(c_2) \cap \text{mid}(t)$, we set $X_t(p) = X_{c_1}(p) \cap X_{c_2}(p)$, otherwise for $p \in (\text{mid}(c_1) \cap \text{mid}(t)) \setminus \text{mid}(c_2)$ we set $X_t(p) = X_{c_1}(p)$, and similarly $X_t(p) = X_{c_2}(p)$ for the remainder of the vertices $p \in (\text{mid}(c_2) \cap \text{mid}(t)) \setminus \text{mid}(c_1)$. Thus we get $N_\eta(t) = \langle X_t(p) \rangle_{p \in \text{mid}(t)}$. Observe that the computed tuple indeed adheres to the definition of a signature.

For every node $t$ with children $c_1, c_2$, we compute the combination $\mathcal{B}(s_1, s_2)$ of all consistent signatures of children $s_1 \in D(c_1), s_2 \in D(c_2)$. We prove that exactly those signatures are the valid signatures of $t$.

**Lemma 10.** *For an internal node $t \in T$ with children $c_1, c_2$, the table $D(t)$ of its valid signatures can be computed using the following recurrence relation:*

$$D(t) = \{\mathcal{B}(s_1, s_2) \mid s_1 \in D(c_1), s_2 \in D(c_2), s_1, s_2 \text{ consistent}\}.$$

*Proof.* Given $c_1, c_2, t \in V(T)$ such that $c_1, c_2$ are the children of $t$, to prove that the

recurrence relation holds, we first show that

$$\{\mathcal{B}(s_1, s_2) \mid s_1 \in D(c_1), s_2 \in D(c_2), s_1, s_2 \text{ consistent}\} \subseteq D(t).$$

Fix two arbitrary signatures $s_1 \in D(c_1), s_2 \in D(c_2)$ that are consistent and let $s_t = \mathcal{B}(s_1, s_2)$. We have to show that $s_t$ is valid, that is, there exists a corresponding partial solution $\sigma'$. In the following, let

$$\begin{aligned}
s_1 &= (S_{\text{in}}(c_1), C_{\text{out}}(c_1), M_{c_1}, N_\eta(c_1)),\\
s_2 &= (S_{\text{in}}(c_2), C_{\text{out}}(c_2), M_{c_2}, N_\eta(c_2)),\\
s_t &= \mathcal{B}(s_1, s_2) = (S_{\text{in}}(t), C_{\text{out}}(t), M_t, N_\eta(t)), \text{ and}\\
\sigma' &= (S'_\curlywedge, (N_{\dot v})_{\dot v \in S'_\curlywedge}, \Gamma') \text{ (if } \sigma' \text{ exists).}
\end{aligned}$$

Since $s_1, s_2$ are valid a few properties are immediately verified: $S'_\curlywedge$, the vertices in $S_{\text{in}}(t)$ and inside the cycles of $C_{\text{out}}(t)$, are a union of the split vertices in each child's signature (where some nesting-graph vertices are moved to $S_{\text{in}}(t)$), thus $S'_\curlywedge \subseteq S_\curlywedge$. Additionally, for every $\dot v \in S'_\curlywedge$, we have $N_{\dot v} \subseteq N_G(\mathsf{orig}(\dot v))$ since all coverage is inherited, and for every $v \in \mathsf{orig}(S'_\curlywedge)$, the family $\{N_{\dot v} \mid \dot v \in \mathsf{copies}(v)\}$ partitions $N_G(v)$. Additionally, while $\mathcal{B}$ omits some nesting graphs, it does not change the graphs themselves, in particular, the edge set induced by $S'_\curlywedge$ still corresponds to the guess made by Branching Rule 2. Thus it is planar.

For each pistil $p$ inside of $\eta(t)$, we want to show that they have no missing neighbors remaining. Since $s_1$ and $s_2$ are valid, all the pistils that were inside $\eta(c_1), \eta(c_2)$ are already covered, we must show that their remaining noose vertices that are inner vertices in $\eta(t)$ do not have remaining missing neighbors. This is done by $\mathcal{B}$ in the third step, for every $v \in I$ (as defined in test 3) where $X_{c_1}(v) \in N_\eta(c_1), X_{c_2}(v) \in N_\eta(c_2)$ we have $X_{c_1}(v) \subseteq N_G(v)$ and $X_{c_2}(v) \subseteq N_G(v)$ (by definition). Since $X_{c_1}(v) \cap X_{c_2}(v) = \emptyset$, by test 2, we know that $(N_G(v) \setminus X_{c_1}(v)) \cup (N_G(v) \setminus X_{c_2}(v)) = N_G(v)$. Thus, the missing neighbors sets for vertices which are not in $\text{mid}(t)$ can be safely removed in $s_t$ as they are empty in the union of the two partial solutions. For the pistil in $\text{mid}(t)$ we must make sure they are missing the correct neighbors. Consider $p \in \text{mid}(t)$; it is either in $\text{mid}(c_1)$, $\text{mid}(c_2)$, or both. If $p$ is only in one child noose, w.l.o.g., assume $p \in \text{mid}(c_1)$. Then $X_{c_1}(p) = X_t(p)$, the missing neighbors are the same as in the child noose and remain for the parent noose. If $p$ is shared by both nooses, then $p$'s missing neighbors are vertices missing in both children nooses which corresponds to what $\mathcal{B}$ computes.

We finally show that we can find a planar drawing $\Gamma'$ that extends $\Gamma$. Such drawings exist for $\eta(c_1)$ and $\eta(c_2)$, we call them $\Gamma_1$ and $\Gamma_2$ respectively, and we show that $\Gamma'$ exists for the subgraph inside $\eta(t)$ and is planar, meaning we now show that we can find the embeddings for the required copies in $\Gamma'$ and connect them without inducing crossings. The solution drawing for the faces not shared between $\eta(c_1)$ and $\eta(c_2)$ can directly be extended to $\Gamma'$ in a planar way: $\mathcal{B}$ verifies in the first test that the only copies shared by both signatures' vertices were necessarily in nesting graphs that correspond

to faces shared by both nooses, which ensures that there are no edges spanning from a face of $\eta(c_1)$ to a face of $\eta(c_2)$ which could create crossings. Next we explain how, for the remaining current faces present in both children nooses, we use the drawings of $\Gamma_1$ and $\Gamma_2$ to create $\Gamma'$. For one such face, there is a set of copies in $\Gamma_1$ that do not appear in $\Gamma_2$, or do not cover vertices in $\Gamma_2$. For those we can replicate the embedding in $\Gamma'$ as it was in $\Gamma_1$. We can do the same with $\Gamma_2$. Consider the copies that cover vertices in both $\Gamma_1$ and $\Gamma_2$; we now show we can also replicate their embedding in $\Gamma'$ without creating crossings. For a vertex $\dot{u}$ with neighbors $u_1 \in G'_1$ and $u_2 \in G'_2$, let us assume it does not have a planar embedding, meaning we find a crossing involving a split copy $\dot{x}$. This means that when traversing $f$, we can find a subsequence of vertices $u_1, x_1, u_2, x_2$ such that $x_1, x_2 \in N_{\dot{x}}$. This can be extended to the vertices of the nesting graph: there must be two neighbors of $\dot{u}$ and two neighbors of $\dot{x}$ on the cycle that are alternating. This means that the two neighborhoods are not sequential in the cyclic ordering of the vertices on the cycle of the nesting graph. This contradicts the validity of the children signatures, and this situation is not possible, showing that we can create $\Gamma'$ by replicating the neighborhoods and embeddings in $\Gamma_1$ and $\Gamma_2$.

Thus $\sigma'$ is a partial solution for $s_t$ and hence $s_t$ is valid.

We now show that $D(t) \subseteq \{\mathcal{B}(s_1, s_2) \mid s_1 \in D(c_1), s_2 \in D(c_2), s_1, s_2 \text{ consistent}\}$. Given a valid signature $s_t \in D(t)$, we show that it is possible to find two valid consistent signatures $s_1 = (S_{\text{in}}(c_1), C_{\text{out}}(c_1), M_{c_1}, N_\eta(c_1))$ for $c_1$ and $s_2 = (S_{\text{in}}(c_2), C_{\text{out}}(c_2), M_{c_2}, N_\eta(c_2))$ for $c_2$. Since $s_t$ is valid there is a corresponding partial solution $\sigma' = (S'_\lambda, (N_{\dot{v}})_{\dot{v} \in S'_\lambda}, \Gamma')$ where $\Gamma'$ is the drawing of the solution graph inside $\eta(t)$. We can find a partial solution for the graphs $G'_1$ and $G'_2$ inside $\eta(c_1)$ and $\eta(c_2)$ respectively from $\sigma'$. To compose the sets ${S'_\lambda}^1$ and ${S'_\lambda}^2$ of copies in the partial solution for $\eta(c_1)$ and $\eta(c_2)$ respectively, we check for each copy in $S'_\lambda$ its neighborhood in $\Gamma'$. If the copy has a neighbor in $G'_1$ is it added to ${S'_\lambda}^1$ and if it covers a pistil in $G'_2$ is it added to ${S'_\lambda}^2$. Thus we obtain partial solutions for $G'_1$ and $G'_2$ using the sets of copies in $S'_{\lambda 1}$ and $S'_{\lambda 2}$ and their embedding in $\Gamma'$.

By definition of the signature of a partial solution, there exists a valid signature $s_1$ and $s_2$ for the two children nodes. We now show that these are consistent:

Since $\Gamma'$ is planar, any vertex embedded in a face $f$ inside $\eta(c_1)$ can only be incident to vertices in $f$. Meaning the first test of $\mathcal{B}$ is passed.

The shared current faces of $s_1$ and $s_2$ will also necessarily have the same nesting graph as they are the same face in $G'_t$: They induce the same nesting graphs, and thus the third test is passed.

As for the second test, since $\sigma'$ is a partial solution, all of the inner vertices in $G'_t$ have no missing neighbors, meaning that the noose vertices of $\eta(c_1)$ and $\eta(c_2)$ that are not noose vertices of $\eta(t)$ are covered by a vertex in a face of $G'_1$ or $G'_2$, which is reflected in the signature of the partial solution. The vertices on the noose may have some neighbors in faces that are not in $G'_t$, those are the missing neighbors in the parent signature.

Lastly, we assume that the mappings created by $\mathcal{B}$ do not pass the fourth test, meaning

without loss of generality in a face $f$'s nesting graph $C_f$ we encounter $p_s^1$ then $p_s^2$ then $p_e^1$ and $p_e^2$. This ordering is strict since the test is not passed. By construction, vertex $p_s^2$ represents either, a pistil in $G'_1$ and $G'_2$, or a path of pistils and non-pistils in $G'_1$ and $G'_2$ that were contracted to a make $p_s^2$ in the nesting graph. In the first case this means that the pistil $p_s^2$ represents a noose pistil and in the second case one of the vertices on the path is on the noose, thus after contraction $p_s^2$ is on the noose. Since once the traversal of $f$ leaves $G'_1$ by passing the noose vertex $p_s^2$, the next encountered pistils are pistils of $G'_2$ and $p_e^1$ cannot be any of the cycle vertex that represents a pistil of $G'_2$, it must be the case that $p_s^2 = p_e^1$, which contradicts our assumption.

Thus $s_1$ and $s_2$ are consistent, the recurrence relation is correct, and Algorithm $\mathcal{B}$ computes all valid signatures. □

We now give the running time for merging the tables of signatures for two child nodes.

**Lemma 11.** *Let $t$ be a node in $T$ that is neither the root nor a leaf and let $c_1, c_2$ be $t$'s children. Given the tables $D(c_1)$ and $D(c_2)$ we can fill the table $D(t)$ in $\mathcal{O}(N_s(k)^2 k^4)$ time.*

*Proof.* Algorithm $\mathcal{B}$ first does four different tests. To verify that the inner vertex sets of size at most $2k$ are disjoint requires quadratic time, we additionally need to compute the vertex sets of the nesting graph which can be done in linear time. Overall the first test requires $\mathcal{O}(k^2)$ time. Checking that the at most $20k$ nesting graphs are equal can be done in linear time for the second test, for a $\mathcal{O}(k)$ runtime. There are at most $20k$ noose vertices which have at most $k$ missing neighbors. We must verify that the missing neighbors sets are disjoint in $\mathcal{O}(k^2)$ time, thus the third test takes $\mathcal{O}(k^3)$ overall. Lastly, a clockwise traversal of the cycle of length at most $4k$ requires linear time. Overall these check are done for each signature pair in the table of valid signatures for the children, which are of size at most $N_s(k)$, the number of possible signatures. Overall, $\mathcal{B}$ can be fill the table for an internal node in $\mathcal{O}(N_s(k)^2 \cdot k^3)$ time. □

This concludes the description of the algorithm for computing the valid signatures for internal nodes.

**Checking Signatures at the Root Node.** Finally, we check for the existence of a solution $\sigma$ by looking at the valid signatures of the children of the root node. In the initialization we split a single edge $e_r = (r_1, r_2)$ into two edges by connecting the end points of $e_r$ to a new root node $r$. As a result $mid(r_1) = mid(r_2) = mid(e_r)$, and the union of the graphs inside $\eta(r_1)$ and $\eta(r_2)$ would make up the whole input graph. With the previous section we have shown how to compute the tables $D(r_1)$ and $D(r_2)$. We now show how given $s_1 \in D(r_1), s_2 \in D(r_2)$ we decide if a solution exists. We reuse here algorithm $\mathcal{B}$ to decide whether the two signatures are consistent. As there are no current faces anymore, the only relevant element in the tuple is the set $S_{\text{in}}$ of vertices that were used to cover the pistils in $G'$. We run one last test on the set $S_\lambda \setminus S_{\text{in}}$. The edge set

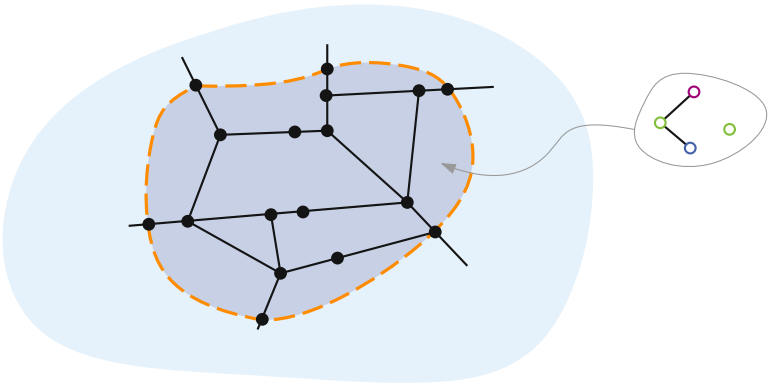


Figure 3.13: For the (orange) noose at the root, $\mathcal{B}$ checks signatures for the light blue and dark blue subgraphs. If a pair of consistent signatures is found, we check if the remaining copies (colored circles) in $S_\lambda$ form a planar graph, that can be embedded in any face of our drawing.

guessed by Branching Rule 2 holds some of the coverage of the split vertices, but if a split vertex is not in $S_{\text{in}}$ and has a neighbor in $G_{S_\lambda}$, then we must be able to realize those last edges. Hence we verify that the graph $G_{S_\lambda}[S_\lambda \setminus S_{\text{in}}]$ is planar, and if so, embed it inside any face of a partial solution. Thus, if this test is passed, we find that a solution $\sigma$ exists.

**Lemma 12.** *For an instance $I$ of Split Set Re-Embedding Decomp with input graph $G'$ and modified sphere-cut decomposition tree $T$ rooted on $r$ where $r_1, r_2$ are the children of $r$ with $G'_1, G'_2$ the subgraphs inside $\eta(r_1), \eta(r_2)$ respectively, such that $G'_1 \cup G'_2 = G'$, $I$ has a solution if and only if there exist valid consistent signatures for $r_1, r_2$.*

*Proof.* Given that two valid consistent signatures $s_1, s_2$ exist, for $r_1, r_2 \in V(T)$ with $G'_1, G'_2$ the subgraphs inside $\eta(r_1), \eta(r_2)$ respectively, such that $G'_1 \cup G'_2 = G'$ we show that $I$ has a solution.

From Lemma 10, the existence of valid consistent signatures $s_1, s_2$ ensures that $D(r) \neq \emptyset$. For $s_r \in D(r)$ s.t. $s_r = (S_{\text{in}}(r), C_{\text{out}}(r), M_r, N_\eta(r))$, $s_r$ is valid, meaning it corresponds to a partial solution $\sigma' = (S'_\lambda, (N_{\dot{v}})_{\dot{v} \in S'_\lambda}, \Gamma')$. We can construct a solution for $I$ that looks as follows: $(S_\lambda, \mathsf{orig}, \mathsf{copies}, (N_{\dot{v}})_{\dot{v} \in S_\lambda}, \Gamma^*)$. The mappings $\mathsf{copies}$ and $\mathsf{orig}$ are those guessed in Branching Rule 1. We know that $S'_\lambda = S_{\text{in}}(r) \subseteq S_\lambda$. The missing vertices $S_\lambda \setminus S'_\lambda$ can be added in any face of the partial solution's drawing $\Gamma'$ to get the drawing $\Gamma^*$. We will argue later that $\Gamma^*$ is still planar.

We now show for $(N_{\dot{v}})_{\dot{v} \in S_\lambda}$, that the family $\{N_{\dot{v}} \mid \dot{v} \in \mathsf{copies}^*(v)\}$ is a partition of $N_G(v)$. First consider a pistil $p \in \eta(r_1)$, since $\text{mid}(r_1) = \text{mid}(r_2)$, $p \in \eta(r_2)$. When creating the set $N_\eta(r)$, $\mathcal{B}$ removes the sets $X_t(p)$ from $N_\eta(r)$ where $p \in \text{mid}(c_1) \cap \text{mid}(c_2)$, hence $N_\eta(r) = \emptyset$. Now we make a case distinction on the different types of pistils. For a pistil $p \in G'$, w.l.o.g. assume $p \in V(G'_1)$ and $p \notin \text{mid}(r_1)$, since $s_1$ is valid, $p$ is covered. If $p \in \text{mid}(r_1)$ then $p \in \text{mid}(r_2)$ and since $N_\eta(r) = \emptyset$, $p$ is covered. For a split vertex

$\dot{p} \in S_{\text{in}}(s_r)$, since $s_r$ is valid, $\dot{p}$ is covered. Lastly, if $\dot{p} \in S_\lambda \setminus S_{\text{in}}(s_r)$, $\dot{p}$ has all of its missing neighbors in $S_\lambda$: $\dot{p}$ cannot have neighbors in $S_{\text{in}}(s_r)$, as vertices in $S_{\text{in}}(s_r)$ are already completely covered. We only need to embed $G_{S_\lambda}[S_\lambda \setminus S_{\text{in}}(s_r)]$ in $\Gamma'$ to obtain the coverage of $\dot{p}$.

Since $s_r$ is valid, $\Gamma'$ is planar. The only vertices of $S_\lambda$ not yet embedded in $\Gamma'$ are the vertices not in $S_\lambda \setminus S_{\text{in}}$. The last test, executed after $\mathcal{B}$, verifies that the graph consisting of the remaining copies $G_{S_\lambda}[S_\lambda \setminus S_{\text{in}}(s_r)]$ is planar, and since it is not a connected component of $G'$, it can be embedded into any of its faces and preserve planarity. Hence, there exists a planar drawing $\Gamma^*$ of the solution.

Given a solution $I = (S_\lambda, \mathsf{orig}, \mathsf{copies}, (N_{\dot{v}})_{\dot{v} \in S_\lambda}, \Gamma^*)$ we now show that we can find two valid consistent signatures signatures $s_1, s_2$ for $r_1, r_2 \in V(T)$ with $G'_1, G'_2$ the subgraphs inside $\eta(r_1), \eta(r_2)$ respectively, such that $G'_1 \cup G'_2 = G'$. By definition, the drawing $\Gamma^*$ is the drawing of a graph extending $G'$, we remove from it any component not connected to $G'$. The vertices we remove are necessarily vertices in $S_\lambda$ as all other vertices are in $G'$. We call $S'_\lambda$ the remaining split vertices and $\Gamma'$ the obtained drawing. Since every pistil in $\Gamma^*$ was covered and we did not remove vertices connected to $G'$, every pistil in $\Gamma'$ is covered, and it is a planar drawing. With these elements we construct $\sigma' = (S'_\lambda, (N_{\dot{v}})_{\dot{v} \in S'_\lambda}, \Gamma')$ a partial solution for $G'$. We can now obtain the signature of a partial solution, $s_r = (S_{\text{in}}(r), C_{\text{out}}(r), M_r, N_\eta(r))$ where $S_\lambda = S_{\text{in}}(r)$ — this is a special case with no current faces. Lemma 10 tells us that given a valid signature for a parent node, there must be two valid consistent signatures for its children, hence, as $r$ corresponds to $G'$, its children must partition $G'$ in two and thus we find two valid consistent signatures signatures $s_1, s_2$ for $r_1, r_2 \in V(T)$ with $G'_1, G'_2$ the subgraphs inside $\eta(r_1), \eta(r_2)$ respectively, such that $G'_1 \cup G'_2 = G'$. □

**Lemma 13.** *Given two valid signature tables $D(r_1)$ and $D(r_2)$ where $\text{mid}(r_1) = \text{mid}(r_2)$ such that the union of the subgraphs inside $\eta(r_1), \eta(r_2)$ is the whole input graph, we can decide in $\mathcal{O}(N_s(k)^2 k^3)$ time if our input is a yes instance.*

*Proof.* Lemma 11 showed that $\mathcal{B}$ requires $\mathcal{O}(N_s(k)^2 k^3)$ time to compute the table for an internal node. On the root level the only difference is the final test that checks the planarity of the graph induced by the at most $2k$ vertices of $S_\lambda$, which are unused in the signature. This can be done in $\mathcal{O}(k)$ time. Thus, we can check for the existence of a solution in $\mathcal{O}(N_s(k)^2 k^3)$ time. □

This now gives us the necessary tools to prove Theorem 5.

*Proof of Theorem 5.* From the input instance of Split Set Re-Embedding, we first find mappings between candidate vertices and copies, and determine how the graph induced by candidate vertices maps to copies. Lemmas 1 and 2 show that an exhaustive search can do this in $\mathcal{O}(2^k k^3)$ time. Next, we show with Corollary 2 and Lemmas 4 and 5 that we can apply transformations to the input graph in linear time to obtain a

graph, whose sphere-cut decomposition we can compute. Thus we obtain an instance of Split Set Re-Embedding Decomp, that we solve using our dynamic programming algorithm.

The dynamic programming algorithm computes partial solutions starting from the leaves of the sphere-cut decomposition. By Lemma 8 we know that all valid signatures for all $m$ leaves can be correctly computed, and this can be done in $\mathcal{O}(N_s(k)k2^{2k}2k!)$ time per leaf (see Lemma 9). The number of possible signatures $N_s(k)$ can be bounded using Lemma 7, to get $N_s(k) = 2^{2k} \cdot 2^{20k^2} \cdot 2^{O(k \log k)} \cdot \binom{8k}{2} = 2^{O(k^2)}$. Those signatures can be enumerated in $\mathcal{O}(N_s(k))$. The dynamic programming now correctly computes all valid signatures for all $m-1$ internal nodes of the sphere-cut decomposition in bottom up fashion (see Lemma 10), in $\mathcal{O}(N_s(k)^2k^3)$ time per internal node, as shown in Lemma 11. Finally, we arrive at the root, where we can correctly determine whether a solution to Split Set Re-Embedding exists, by Lemma 12. This takes an additional $\mathcal{O}(N_s(k)^2k^3)$ time (see Lemma 13). Thus, we can solve any instance in $2^{O(k^2)} \cdot n^{O(1)}$ time □

## 3.5 Chapter Conclusion

In this chapter, we introduced the embedded splitting number problem for planarizing graph drawings by vertex splitting and showed its NP-completeness. However, fixed-parameter tractability is only established for the Split Set Re-Embedding subproblem, the parameterized complexity of Embedded Splitting Number remains open.

A trivial XP-algorithm for Embedded Splitting Number can provide appropriate inputs to Split Set Re-Embedding as follows: check for any subset of up to $k$ vertices, whether removing those vertices results in a planar input drawing, and branch on all such subsets.

An FPT-algorithm for Embedded Splitting Number should similarly select vertices whose removal results in a planar drawing. However, this is not sufficient, as it may be necessary to also consider vertices whose removal merges two faces $f_1$ and $f_2$ into $f_3$. Thus less copies may be required to cover pistils in $f_3$, than placing copies in both $f_1$ and $f_2$.

Furthermore, many variations of embedded splitting number are interesting for future work. For example, rather than aiming for planarity, we can utilize vertex splitting for crossing minimization. Other possible extensions can adapt the splitting operation. Vertex explosion (or splitting a vertex degree many times) behaves similarly to minimizing the number of vertices split at least once, and both problems in this setting are equivalent to Embedded Vertex Deletion. But, since the split operation allows both creating an additional copy of a vertex and re-embedding it, one could consider alternatives in which changing a the embedding of a vertex is a separate, cheaper operation.

CHAPTER 4

# Planar Drawings

In Chapter 3 we saw that splitting graph drawings to planarity is NP-complete. We now continue the investigation of the vertex splitting operation when applied to restricted graph classes, beginning here with splitting drawings of planar graphs to outerplanarity. This chapter is based on joint work with Martin Gronemann and Martin Nöllenburg. This paper was presented at WALCOM 2023 [GNV23] and invited to the corresponding JGAA special issue.

Graph editing problems are fundamental problems in graph theory. They define a set of basic operations on a graph $G$ and ask for the minimum number of these operations necessary in order to turn $G$ into a graph of a desired target graph class $\mathcal{G}$ [NSS01, LY80, Yan78, Kan96]. For instance, in the Cluster Editing problem [SST04] the operations are insertions or deletions of individual edges and the target graph class are cluster graphs, i.e., unions of vertex-disjoint cliques. In graph drawing, a particularly interesting graph class are planar graphs, for which several related graph editing problems have been studied, e.g., how many vertex deletions are needed to turn an arbitrary graph into a planar one [MS12] or how many vertex splits are needed to obtain a planar graph [JR84, FdFdMN01]. In this chapter, we further this study of vertex splitting, by applied to splitting plane drawings to outerplane ones.

Further, we are translating the graph editing problem into a more geometric or topological drawing editing problem. This means that rather than having complete freedom to choose the neighborhoods and the embeddings of our copies, we consider the existing embedding of the planar embedded, or *plane*, graph. In a plane graph, each vertex has an induced cyclic order of incident edges, which needs to be respected by any vertex split: we must split the vertex's cyclic order into two contiguous intervals, one for each of the two copies. From a different perspective, the two faces that serve as the separators of these two edge intervals are actually merged into a single face by the vertex split.

(a) (b)

Figure 4.1: (a) An instance of Outerplane Splitting Number, where the colored vertices will be split; (b) resulting outerplane graph after the minimum 3 splits.

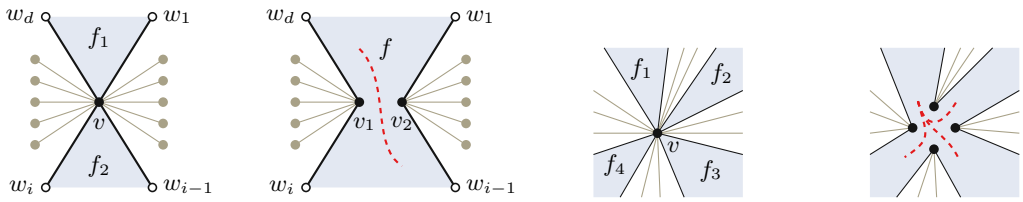
To obtain an outerplanar graphs through vertex splitting operations, we want to apply a minimum number of vertex splits to a plane graph $G$. These splits merge a minimum number of faces in order to obtain an outerplanar embedded graph $G'$, where all vertices are incident to a single face, called the *outer face*. We denote this minimum number of splits as the *outerplane splitting number* $\mathrm{osn}(G)$ of $G$ (see Fig. 4.1). Outerplanar graphs are a prominent graph class in graph drawing (see, e.g., [Bie11, Fra22, LLL19, LL96]) as well as in graph theory and graph algorithms more generally (e.g., [BF02, Fre96, MZ99]). For instance, outerplanar graphs admit planar circular layouts or 1-page book embeddings [BK79]. Additionally, outerplanar graphs often serve as a simpler subclass of planar graphs with good algorithmic properties. For example, they have treewidth 2 and their generalizations to $k$-outerplanar graphs still have bounded treewidth [Bod98, Bie15], which allows for polynomial-time algorithms for NP-complete problems that are tractable for such bounded-treewidth graphs. This, in turn, can be used to obtain a PTAS for these problems on planar graphs [Bak94].

We are now ready to define our main computational problem as follows.

**Problem 6** (Outerplane Splitting Number)**.** Given a plane biconnected graph $G = (V, E)$ and an integer $k$, can we transform $G$ into an outerplane graph $G'$ by applying at most $k$ vertex splits to $G$?

While splitting numbers have been studied mostly for abstract (non-planar) graphs with the goal of turning them into planar graphs, there also exist applied work in graph drawing that make use of vertex splitting to untangle edges [WNV20] or to improve layout quality for community exploration [ADM+22, HBF08]. Regarding vertex splitting for achieving graph properties other than planarity, Trotter and Harary [TH79] studied vertex splitting to turn a graph into an interval graph. Paik et al. [PRS98] considered vertex splitting to remove long paths in directed acyclic graphs and Abu-Khzam et al. [ABFS21] studied heuristics using vertex splitting for a cluster editing problem.

**Structure of the chapter:** We start by showing the key property for our subsequent results, namely that (minimum) sets of vertex splits to turn a plane biconnected graph $G$ into an outerplane one correspond to (minimum) connected face covers in $G$ (Section 4.2),




(a) (b) (c) (d)

Figure 4.2: (a) Two touching faces $f_1, f_2$ with a common vertex $v$ on their boundary. (b) Result of the split of $v$ with respect to $f_1, f_2$ joining them into a new face $f$. (c-d) Merging 4 faces $f_1, \ldots, f_4$ covering a single vertex $v$ with 3 splits.

which in turn are equivalent to (minimum) feedback vertex sets in the dual graph of $G$. Using these equivalences, we then show that for general plane biconnected graphs Outerplane Splitting Number is NP-complete (Section 4.3), whereas for maximal planar graphs we can solve it in polynomial time (Section 4.4). Then, we provide upper and lower bounds on the outerplane splitting number for maximal planar graphs (Section 4.5). Finally, we introduce a SAT formulation to solve a generalized formulation of Outerplane Splitting Number.

## 4.1 Preliminaries

The key concept of our approach is to merge a set of faces of a given plane graph $G = (V, E)$ with vertex set $V = V(G)$ and edge set $E = E(G)$ into one big face which is incident to all vertices of $G$. Hence, the result is outerplanar. The idea is that if two faces $f_1$ and $f_2$ share a vertex $v$ on their boundary (we say $f_1$ and $f_2$ *touch*, see Fig. 4.2a), then we can split $v$ into two new vertices $v_1, v_2$. In this way, we are able to create a narrow gap, which merges $f_1, f_2$ into a bigger face $f$ (see Fig. 4.2b). With this in mind, we formally define an *embedding-preserving split* of a vertex $v$ with regards to two incident faces $f_1$ and $f_2$. We construct a new plane graph $G' = (V', E')$ with $V' = V \setminus \{v\} \cup \{v_1, v_2\}$. Consider the two neighbors of $v$ both incident to $f_1$ and let $w_1$ be the second neighbor in clockwise order. Similarly, let $w_i$ be the second vertex adjacent to $v$ and incident to $f_2$. We call $w_d$ the vertex preceding $w_1$ in the cyclic ordering or the neighbors, with $d$ being the degree of $v$, see Fig. 4.2a. Note that while $w_1 = w_{i-1}$ and $w_i = w_d$ is possible, $w_d \neq w_1$ and $w_{i-1} \neq w_i$. For the set of edges, we now set $E' = E \setminus \{(v, w_1), \ldots, (v, w_d)\} \cup \{(v_2, w_1), \ldots, (v_2, w_{i-1})\} \cup \{(v_1, w_i), \ldots, (v_1, w_d)\}$ and assume that they inherit their embedding from $G$. From now on we refer to this operation simply as a *split* or when $f_1, f_2$ are clear from the context, we may refer to *merging* the two faces at $v$. The vertices $v_1, v_2$ introduced in place of $v$ are called *copies* of $v$. If a copy $v_i$ of a vertex $v$ is split again, then any copy of $v_i$ is also called a copy of the original vertex $v$.

We can now reformulate the task of using as few splits as possible. Our objective is to find a set of faces $S$ that satisfies two conditions. (1) Every vertex in $G$ has to be on the

boundary of at least one face $f \in S$, that is, the faces in $S$ *cover* all vertices in $V$.[1] And (2) for every two faces $f, f' \in S$ there exists a set of faces $\{f_1, \ldots, f_k\} \subseteq S$ such that $f = f_1, \ldots, f_k = f'$, and $f_i$ touches $f_{i+1}$ for $1 \leq i < k$. In other words, $S$ is connected in terms of touching faces. We now introduce the main tool in our constructions that formalizes this concept.

## 4.2 Face-Vertex Incidence Graph

Let $G = (V, E)$ be a plane biconnected graph and $F$ its set of faces. The *face-vertex incidence graph* is defined as $H = (V \cup F, E_H)$ and contains the edges $E_H = \{(v, f) \in V \times F : v \text{ is on the boundary of } f\}$. Graph $H$ is by construction bipartite and we assume that it is plane by placing each vertex $f \in F$ into its corresponding face in $G$.

**Definition 4.** Let $G$ be a plane biconnected graph, let $F$ be the set of faces of $G$, and let $H$ be its face-vertex incidence graph. A *face cover* of $G$ is a set $S \subseteq F$ of faces such that every vertex $v \in V$ is incident to at least one face in $S$. A face cover $S$ of $G$ is a *connected* face cover if the induced subgraph $H[S \cup V]$ of $S \cup V$ in $H$ is connected.

We point out that the problem of finding a connected face cover is not equivalent to the Connected Face Hitting Set Problem [SS10], where a connected set of vertices incident to every face is computed. We continue with two lemmas that are concerned with merging multiple faces at the same vertex (Fig. 4.2c).

**Lemma 14.** *Let $G$ be a plane biconnected graph and $S \subseteq F$ a subset of the faces $F$ of $G$ that all have the vertex $v \in V$ on their boundary. Then $|S| - 1$ splits are sufficient to merge the faces of $S$ into one.*

*Proof.* Let $f_1, \ldots, f_k$ with $k = |S|$ be the faces of $S$ in the clockwise order as they appear around $v$ ($f_1$ chosen arbitrarily). We iteratively merge $f_1$ with $f_i$ for $2 \leq i \leq k$, which requires in total $|S| - 1$ splits (see Fig. 4.2c and Fig. 4.2d). □

**Lemma 15.** *Let $G$ be a plane biconnected graph and let $S$ be a connected face cover of $G$. Then $|S| - 1$ splits are sufficient to merge the faces of $S$ into one.*

*Proof.* Let $H' = H[S \cup V]$ and compute a spanning tree $T$ in $H'$. For every vertex $v \in V(T) \cap V(G)$, we apply Lemma 14 with the face set $F'(v) = \{f \in S \cap V(T) \mid (v, f) \in E(T)\}$. We root the tree at an arbitrary face $f' \in S$, which provides a hierarchy on the vertices and faces in $T$. Every vertex $v \in V(T) \cap V(G)$ requires by Lemma 14 $|F'(v)| - 1$ splits. Note that that for all leaf vertices in $T$, $|F'(v)| = 1$, i.e., they will not be split. Each split is charged to the children of $v$ in $T$. Since $H$ is bipartite, so is $T$. It follows that every face $f \in S \setminus \{f'\}$ is charged exactly once by its parent, thus $|S| - 1$ splits suffice. □

[1] Testing whether such $S$ with $|S| \leq k$ exists, is the NP-complete problem Face Cover [BM88].

**Lemma 16.** *Let $G$ be a plane biconnected graph and $\sigma$ a sequence of $k$ splits to make $G$ outerplane. Then $G$ has a connected face cover of size $k+1$.*

*Proof.* Since by definition applying $\sigma$ to $G$ creates a single big face that is incident to all vertices in $V(G)$ by iteratively merging pairs of original faces defining a set $S \subseteq F$, it is clear that $S$ is a face cover of $G$ and since the result of the vertex splits and face merges creates a single face, set $S$ must also be connected. □

As a consequence of Lemmas 15 and 16 we obtain that Outerplane Splitting Number and computing a minimum connected face cover are equivalent.

**Theorem 6.** *Let $G$ be a plane biconnected graph. Then $G$ has outerplane splitting number $k$ if and only if it has a connected face cover of size $k+1$.*

## 4.3 NP-completeness

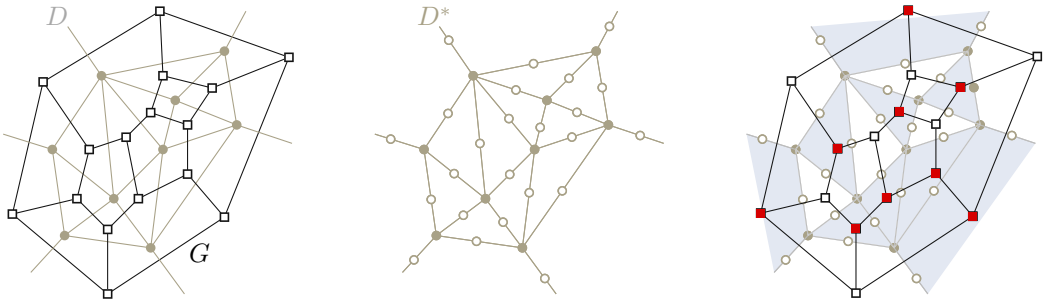
In this section, we prove that finding a connected face cover of size $k$ (and thus Outerplane Splitting Number) is NP-complete. The idea is to take the dual of a planar biconnected Vertex Cover instance and subdivide every edge once (we call this an *all-1-subdivision*). Note that the all-1-subdivision of a graph $G$ corresponds to its vertex-edge incidence graph and the all-1-subdivision of the dual of $G$ corresponds to the face-edge incidence graph of $G$. A connected face cover then corresponds to a vertex cover in the original graph, and vice versa. The following property greatly simplifies the arguments regarding Definition 4.

**Property 1.** Let $G'$ be an all-1-subdivision of a biconnected planar graph $G$ and $S$ a set of faces that cover $V(G')$. Then $S$ is a connected face cover of $G'$.

*Proof.* Let $H$ be the all-1-subdivision of the dual of $G$, and assume to the contrary that the induced subgraph $H' = H[S \cup V(G)]$ is not connected. Then there exists an edge $(u, v) \in E(G)$ such that $u$ and $v$ are in different connected components in $H'$. Let $w$ be the subdivision vertex of $(u, v)$ in $G'$. As a subdivision vertex, $w$ is incident to only two faces, one of which, say $f$, must be contained in $S$. But $f$ is also incident to $u$ and $v$ and hence $u$ and $v$ are in the same component of $H'$ via face $f$, a contradiction. Hence $H'$ is connected and $S$ is a connected face cover of $G'$. □

The proof of the next theorem is very similar to the reduction of Bienstock and Monma to show NP-completeness of Face Cover [BM88]; due to differences in the problem definitions, such as the connectivity of the face cover and whether the input graph is plane or not, we provide the full reduction for the sake of completeness.

**Theorem 7.** *Deciding whether a plane biconnected graph $G$ has a connected face cover of size at most $k$ is NP-complete.*




(a) primal (black) and dual graph (gray)

(b) subdivision of the dual graph

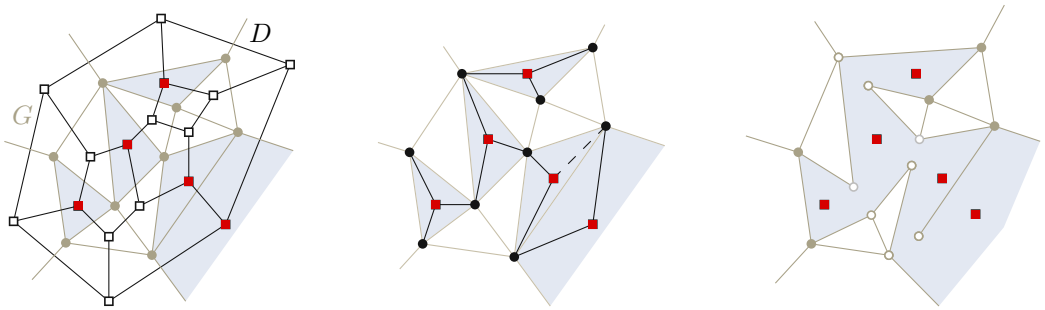
(c) vertex cover (red) in the primal graph

Figure 4.3: Link between the primal graph $G$, its vertex cover, the dual $D$ and its subdivision $D^*$.

*Proof.* Clearly the problem is in NP. To prove hardness, we first introduce some notation. Let $G$ be a plane biconnected graph and $D$ the corresponding dual graph. Furthermore, let $D^*$ be the all-1-subdivision of $D$. We prove now that a connected face cover $S^*$ of size $k$ in $D^*$ is in a one-to-one correspondence with a vertex cover $S$ of size $k$ in $G$ (see Fig. 4.3). More specifically, we show that the dual vertices of the faces of $S^*$ that form a connected face cover in $D^*$, are a vertex cover for $G$ and vice versa. The reduction is from the NP-complete Vertex Cover problem in biconnected planar graphs in which all vertices have degree 3 (cubic graphs) [Moh01].

**Connected Face Cover ⇒ Vertex Cover:** Let $G$ be such a biconnected plane Vertex Cover instance. Assume we have a connected face cover $S^*$ with $|S^*| = k$ for $D^*$. Note that the faces of $D^*$ correspond to the vertices in $G$. We claim that the faces $S^*$, when mapped to the corresponding vertices $S \subseteq V(G)$ are a vertex cover for $G$. Assume otherwise, that is, there exists an edge $e^* \in E(G)$ that has no endpoints in $S$. However, $e^*$ has a dual edge $e \in E(D)$ and therefore a subdivision vertex $v_e \in V(D^*)$. Hence, there is a face $f \in S^*$ that has $v_e$ on its boundary by definition of connected face cover. And when mapped to $D$, $f$ has $e$ on its boundary, which implies that the primal edge $e^*$ has at least one endpoint in $S^*$; a contradiction.

**Vertex Cover ⇒ Connected Face Cover:** To prove that a vertex cover $S$ induces a connected face cover $S^*$ in $D^*$, we have to prove that $S^*$ covers all vertices and the induced subgraph in the face-vertex incidence graph $H$ is connected. We proceed as in the other direction. $S$ covers all edges in $E(G)$, thus every edge $e \in E(D)$ is bounded by at least one face of $S^*$. Hence, every subdivision vertex in $V(D^*)$ is covered by a face of $S^*$. Furthermore, every vertex in $D^*$ is adjacent to a subdivision vertex, thus, also covered by a face in $S^*$. Since $S^*$ is covering all vertices, we obtain from Property 1 that $S^*$ is a connected face cover. □




(a) primal (gray) and dual graph (black)

(b) spanning tree (black) of the connected face cover

(c) outerplanar graph after splitting along the tree

Figure 4.4: The connected face cover (blue) is a feedback vertex set (red) in the dual.

## 4.4 Feedback Vertex Set Approach

A *feedback vertex set* $S^\circ \subset V(G)$ of a graph $G$ is a vertex subset such that the induced subgraph $G[V(G) \setminus S^\circ]$ is acyclic. We show here that finding a connected face cover $S$ of size $k$ for a plane biconnected graph $G$ is equivalent to finding a feedback vertex set $S^\circ \subset V(D)$ of size $k$ in the dual graph $D$ of $G$. The *weak dual*, i.e., the dual without a vertex for the outer face, of an outerplanar graph is a forest. Thus we must find the smallest number of splits in $G$ which transform $D$ into a forest. In other words, we must break all the cycles in $D$, and hence all of the vertices in the feedback vertex set $S^\circ$ of $D$ correspond to the faces of $G$ that should be merged together (see Fig. 4.4).

**Property 2.** Let $H$ be the face-vertex incidence graph of a plane biconnected graph $G$ and let $S^\circ$ be a feedback vertex set in the dual $D$ of $G$. Then $S^\circ$ induces a connected face cover $S$ in $G$.

*Proof.* We need to show that $S^\circ$ is a face cover and that it is connected. First, assume there is a vertex $v \in V(G)$ of degree $\deg(v) = d$ that is not incident to a vertex in $S^\circ$, i.e., a face of $G$. Since $G$ is biconnected, $v$ is incident to $d$ faces $f_1, \ldots, f_d$, none of which is contained in $S^\circ$. But then $D[V(D) \setminus S^\circ]$ has a cycle $(f_1, \ldots, f_d)$, a contradiction.

Next, we define $\overline{S^\circ} = V(D) \setminus S^\circ$ as the complement of the feedback vertex set $S^\circ$ in $D$. Assume that $H[V \cup S^\circ]$ has at least two separate connected components $C_1, C_2$. Then there must exist a closed curve in the plane separating $C_1$ from $C_2$, which avoids the faces in $S^\circ$ and instead passes through a sequence $(f_1, \ldots, f_\ell)$ of faces in $\overline{S^\circ}$, where each pair $(f_i, f_{i+1})$ for $i \in \{1, \ldots, \ell - 1\}$ as well as $(f_\ell, f_1)$ are adjacent in the dual $D$. Again this implies that there is a cycle in $D[V(D) \setminus S^\circ]$, a contradiction. Thus $S^\circ$ is a connected face cover. □

**Theorem 8.** *A plane biconnected graph $G$ has outerplane splitting number $k$ if and only if its dual $D$ has a minimum feedback vertex set of size $k + 1$.*

*Proof.* Let $S^\circ$ be a minimum feedback vertex set of the dual $D$ of $G$ with cardinality $|S^\circ| = k+1$ and let $H$ be the face-vertex incidence graph of $G$. We know from Property 2 that $H' = H[V(G) \cup S^\circ]$ is connected and hence $S^\circ$ induces a connected face cover $S$ with $|S| = k+1$. Then by Lemma 15 $G$ has $\mathrm{osn}(G) \leq k$.

Let conversely $\sigma$ be a sequence of $k$ vertex splits that turn $G$ into an outerplane graph $G'$ and let $F$ be the set of faces of $G$. By Lemma 16 we obtain a connected face cover $S$ of size $k+1$ consisting of all faces that are merged by $\sigma$. The complement $\overline{S} = F \setminus S$ consists of all faces of $G$ that are not merged by the splits in $\sigma$ and thus are the remaining (inner) faces of the outerplane graph $G'$. Since $G'$ is outerplane and biconnected, $\overline{S}$ is the vertex set of the weak dual of $G'$, which must be a tree. Hence $S$ is a feedback vertex set in $D$ of size $k+1$ and the minimum feedback vertex set in $D$ has size at most $k+1$. □

Since all faces in a maximal planar graph are triangles, the maximum vertex degree of its dual is 3. Thus, we can apply the polynomial-time algorithm of Ueno et al. [UKG88] to this dual, which computes the minimum feedback vertex set in graphs of maximum degree 3 by reducing the instance to polynomial-solvable matroid parity problem instance, and obtain

**Corollary 3.** We can solve Outerplane Splitting Number for maximal planar graphs in polynomial time.

Many other existing results for feedback vertex set extend to Outerplane Splitting Number, e.g., it has a kernel of size $13k$ [BK16] and admits a PTAS [DH05].

## 4.5 Lower and Upper Bounds

In this section we provide some upper and lower bounds on the outerplane splitting number in certain maximal planar graphs.

### 4.5.1 Upper Bounds

Based on the equivalence of Theorem 8 we obtain upper bounds on the outerplane splitting number from suitable upper bounds on the feedback vertex set problem, which has been studied for many graph classes, among them cubic graphs [BHS87]. Liu and Zhao [LZ96] showed that cubic graphs $G = (V, E)$ of girth at least four (resp., three) have a minimum feedback vertex set of size at most $\frac{|V|}{3}$ (resp., $\frac{3|V|}{8}$). Kelly and Liu [KL17] showed that connected planar subcubic graphs of girth at least five have a minimum feedback vertex set of size at most $\frac{2|V|+2}{7}$. Recall that the girth of a graph is the length of its shortest cycle.

**Proposition 2.** The outerplane splitting number of a maximal planar graph $G = (V, E)$ of minimum degree (i) 3, (ii) 4, and (iii) 5, respectively, is at most (i) $\frac{3|V|-10}{4}$, (ii) $\frac{2|V|-7}{3}$, and (iii) $\frac{4|V|-13}{7}$, respectively.

*Proof.* Maximal planar graphs with $n = |V|$ vertices have $2n - 4$ faces. So the corresponding dual graphs have $2n - 4$ vertices. Moreover, since the degree of a vertex in $G$ corresponds to the length of a facial cycle in the dual, graphs with minimum vertex degree 3, 4, or 5 have duals with girth 3, 4, or 5, respectively. So if the minimum degree in $G$ is 3, we obtain an upper bound on the feedback vertex set of $(3n - 6)/4$; if the minimum degree is 4, the bound is $(2n - 4)/3$; and if the minimum degree is 5, the bound is $(4n - 6)/7$. The claim then follows from Theorem 8. □

### 4.5.2 Lower Bounds

We first provide a generic lower bound for the outerplane splitting number of maximal planar graphs. Let $G$ be an $n$-vertex maximal planar graph with $2n - 4$ faces. Each face is a triangle incident to three vertices. In a minimum-size connected face cover $S^*$, the first face covers three vertices. Due to the connectivity requirement, all other faces can add at most two newly covered vertices. Hence we need at least $\frac{n-1}{2}$ faces in any connected face cover. By Theorem 6 this implies that $\mathrm{osn}(G) \geq \frac{n-3}{2}$.

**Proposition 3.** Any maximal planar graph $G$ has outerplane splitting number at least $\frac{|V(G)|-3}{2}$.

Next, towards a better bound, we define a family of maximal planar graphs $T_d = (V_d, E_d)$ of girth 3 for $d \geq 0$ that have outerplane splitting number at least $\frac{2|V_d|-8}{3}$. The family are the complete planar 3-trees of depth $d$, which are defined recursively as follows. The graph $T_0$ is the 4-clique $K_4$. To obtain $T_d$ from $T_{d-1}$ for $d \geq 1$ we subdivide each inner triangular face of $T_{d-1}$ into three triangles by inserting a new vertex and connecting it to the three vertices on the boundary of the face.

**Proposition 4.** The complete planar 3-tree $T_d$ of depth $d$ has outerplane splitting number at least $\frac{2|V_d|-8}{3}$.

*Proof.* Each $T_d$ is a maximal planar graph with $n_d = 3 + \sum_{i=0}^{d} 3^i = \frac{3^{d+1}+5}{2}$ vertices. All $3^d$ leaf-level vertices added into the triangular faces of $T_{d-1}$ in the last step of the construction have degree 3 and are incident to three exclusive faces, i.e., there is no face that covers more than one of these leaf-level vertices. This immediately implies that any face cover of $T_d$, connected or not, has size at least $3^d$. From $n_d = \frac{3^{d+1}+5}{2}$ we obtain $d = \log_3 \frac{2n_d-5}{3}$ and $3^d = \frac{2n_d-5}{3}$. Theorem 6 then implies that $\mathrm{osn}(T_d) \geq \frac{2n_d-8}{3}$. □

## 4.6 SAT Formulation

We showed previously that Outerplane Splitting Number is NP-complete for plane biconnected graphs, we introduce here a SAT formulation that solves the problem, as well as more general instances. Specifically, given a simple graph $G$ and an integer $k$, the SAT formula decides if $G$ can be transformed into an outerplanar graph $G^*$ with at most

$k$ vertex splitting operation. This formula is a weighted Max-SAT formula, meaning each clause gadget is weighted, and the goal is to maximize the sum of the weights of the satisfied clauses. Our formulation is based on a SAT formulation for book embeddings, similar to the one proposed by Bekos et al. [BKZ15], specifically to model crossings and transitivity. As outerplanar graphs are exactly the classes of graphs that can be embedded into 1-page book embeddings, our formulation aims, for the given input graph, to find a set of splits which would allow for the graph to be embedded on a single page.

Intuitively, we build a graph $G' = (V', E')$, where every vertex is represented $k+1$ times, intuitively, one representative is the vertex itself, and the $k$ others represent its potential $k$ copies. For every edge $(u, v)$, we create a complete bipartite graph with one bipartition being all the vertices representing $u$ and the second bipartition being the vertices that represent $v$. We then find a smallest outerplanar subgraph $G^*$ in $G'$ such that every edge and every vertex is represented at least once in the subgraph.

The formula $\mathcal{F}(G, k)$ is written using the conjunctive normal form, and we now introduce the different variables needed to build our constraints. For each vertex $v \in V$ and each copy in $c_1, \ldots, c_k$, the variable $\texttt{vertex}(v, c_i)$ has value 1 if the $i$-th copy is a copy of vertex $v$ and it appears in $G^*$. For all $v \in V$, $\texttt{vertex}(v, 0)$ corresponds to the input vertex and necessarily, $\texttt{vertex}(v, 0) = 1$. Since edge $(u, v) \in E$ can be represented in the final drawing by any combination of copies of $u$ and $v$, we create an edge from every copy of $u$ to every copy of $v$, thus for $0 \geq i, j \geq k$, we create a variable $\texttt{edge}((u, c_i), (v, c_j))$, and the value is set to 1 if the edge is in $G^*$. For two edges $e$ and $e'$, we create the variable $\texttt{coexist}(e, e')$, such that $\texttt{coexist}(e, e') = 1$ if both $e$ and $e'$ are in $G^*$. Lastly, we need a variable to represent the vertex ordering along the outer face (the spine of the book embedding), specifically, for every $u, v \in V$ and $0 \geq i, j \geq k$, we create $\texttt{before}((u, c_i), (v, c_j))$ and set the value to 1 if the $i$-th copy of $u$ appears ahead of the $j$-th copy of $v$ in $G^*$.

We now introduce the set of constraints of $\mathcal{F}(G, k)$. For clarity, we simplify where possible the notation of vertices, and use $v$ to refer to a copy of a vertex in $V$ (instead of $(v, c_i)$). This set of vertices and their copies is called $V'$. Similarly, we sometimes call $e$ an edge between two vertices of $V'$, and denote this edge set with $E'$.

To fix the linear ordering of the vertices on the outer face, we require transitivity constraints. Therefore for $u, v, w \in V'$, three distinct vertices, we create the two following clauses:

$$(\neg\texttt{before}(u, v) \vee \neg\texttt{before}(v, w) \vee \texttt{before}(u, w)) \tag{4.1}$$

$$(\texttt{before}(u, v) \vee \texttt{before}(v, w) \vee \neg\texttt{before}(u, w)) \tag{4.2}$$

which respectively ensure that if $u$ is before $v$ and $v$ is before $w$, then $u$ is before $w$, and that if $u$ is after $v$ and $v$ after $w$, than $u$ is after $w$.

To minimize the number of copies used, for each vertex $v \in V$ and each $1 \geq i \geq k$ we create the following soft constraint (note that for $i = 0$ this constraint does not exist as

the input vertices are kept "for free"):

$$(\neg \texttt{vertex}(v, c_i)) \tag{4.3}$$

meaning that for every non input vertex $v$ that is not split, this clause is satisfied and its weight is added to the objective function.

As we have a budget of $k$ splitting operations, and every splitting operation is done on a single vertex, for every value of $1 \geq i \geq k$, only one vertex can correspond to the $i$th split, hence for two distinct vertices $u, v \in V$:

$$(\neg \texttt{vertex}(u, c_i) \vee \neg \texttt{vertex}(v, c_i)) \tag{4.4}$$

which ensures that for at most one $v \in V$, $\texttt{vertex}(v, c_i) = 1$.

Since for $(u, v) \in E$ we create an edge between all the possible copies of $u$ and $v$, we have to ensure that at least one edge is present in $G^*$ between $u$ or a copy of $u$, and $v$ or a copy of $v$, thus:

$$\begin{gathered}(\texttt{edge}((u, c_0), (v, c_0)) \vee \cdots \vee \texttt{edge}((u, c_0), (v, c_k)) \vee \\ \cdots \\ \vee \texttt{edge}((u, c_k), (v, c_0)) \vee \cdots \vee \texttt{edge}((u, c_k), (v, c_k)))\end{gathered} \tag{4.5}$$

and, for at least one value of $i$ and $j$ such that $0 \geq i, j \geq k$, $\texttt{edge}((u, c_i), (v, c_j)) = 1$.

Most importantly, we need to ensure no two pair of active edges are crossing. To obtain this we first need to know which edge pairs are coexisting in $G^*$ and can thus potentially intersect, for $e, e' \in E'$:

$$(\texttt{coexist}(e, e') \vee \neg \texttt{edge}(e) \vee \neg \texttt{edge}(e')) \tag{4.6}$$

where $\texttt{coexist}(e, e') = 1$ if and only if $\texttt{edge}(e) = \texttt{edge}(e') = 1$.

This allows us to build the crossing constraints, thus for $e, e' \in E'$ such that $e = (u, v)$ and $e' = (s, t)$, we need to characterize all possible vertex orderings that can induce crossing, which forbids the two edges from coexisting, we set $L = \neg \texttt{coexist}(e, e')$:

$$(L \vee \neg \texttt{before}(s, u) \vee \neg \texttt{before}(u, t) \vee \neg \texttt{before}(s, v) \vee \texttt{before}(v, t)) \tag{4.7}$$
$$(L \vee \neg \texttt{before}(s, u) \vee \neg \texttt{before}(u, t) \vee \texttt{before}(s, v) \vee \neg \texttt{before}(v, t)) \tag{4.8}$$
$$(L \vee \neg \texttt{before}(s, u) \vee \texttt{before}(u, t) \vee \neg \texttt{before}(s, v) \vee \neg \texttt{before}(v, t)) \tag{4.9}$$
$$(L \vee \texttt{before}(s, u) \vee \neg \texttt{before}(u, t) \vee \neg \texttt{before}(s, v) \vee \neg \texttt{before}(v, t)) \tag{4.10}$$
$$(L \vee \neg \texttt{before}(s, u) \vee \texttt{before}(u, t) \vee \texttt{before}(s, v) \vee \texttt{before}(v, t)) \tag{4.11}$$
$$(L \vee \texttt{before}(s, u) \vee \neg \texttt{before}(u, t) \vee \texttt{before}(s, v) \vee \texttt{before}(v, t)) \tag{4.12}$$
$$(L \vee \texttt{before}(s, u) \vee \texttt{before}(u, t) \vee \neg \texttt{before}(s, v) \vee \texttt{before}(v, t)) \tag{4.13}$$
$$(L \vee \texttt{before}(s, u) \vee \texttt{before}(u, t) \vee \texttt{before}(s, v) \vee \neg \texttt{before}(v, t)) \tag{4.14}$$

consider Eq. 4.14, if $\texttt{before}(s,u) = \texttt{before}(u,t) = \texttt{before}(s,v) = \neg\texttt{before}(v,t) = 0$, then the corresponding vertex ordering on the spine is in order $t, u, s$ and $v$ which induces a crossing, thus, $\texttt{coexist}(e,e') = 0$ as these two edges cannot coexist in a planar drawing.

Lastly, we can only allow edges in $G^*$ if their endpoints are also in $G^*$, thus for $(u,v) \in E'$,

$$(\neg\texttt{edge}(u,v) \vee \texttt{vertex}(u)) \wedge (\neg\texttt{edge}(u,v) \vee \texttt{vertex}(v))) \tag{4.15}$$

which ensures that $\texttt{edge}(u,v) = 1$ if and only if $\texttt{vertex}(u) = \texttt{vertex}(v) = 1$.

Only the clause that tracks an active copy is a weighted (here soft) clause, every other clause is a hard clause. Since every edge $e \in E$ has $(k+1)^2$ representatives in $E'$, and since the coexistence variables are created for each pairwise edge representative, there are $\mathcal{O}(|E|^3)$ variables and $\mathcal{O}(|E|^6)$ constraints, which makes the current formulation impractical. Using the MaxHS solver [DB], with $G = K_{10}$, and $k = 7$, it took 1min30s to create the constraints, which came up to 35641240 clauses and could not be handled by the solver. On a small instance like $G = K_5$ and $k = 2$, both creating and solving the formula was instantaneous. The usefulness of solving smaller instances is still relevant for graph visualization but the current formulation would need to be improved, or formulated with integer linear programming.

## 4.7 Chapter Conclusion

In this chapter, we have introduced the Outerplane Splitting Number problem and established its complexity for plane biconnected graphs. We found that it is polynomial time solvable for maximal planar graphs, on that many results pertaining to Feedback Vertex Set can naturally be extended to Outerplane Splitting Number.

The main open question remaining revolves around the embedding requirement. Mainly, splitting operations can be carried without retaining the graph topology at all, and placing the copies freely in the drawing.

In general, it is also of interest to understand how the problem differs when the input graph does not have an embedding at all, as in the original splitting number problem. While splitting number has been studied for abstract graphs, vertex splitting to outerplanarity for abstract general graphs has not yet been studied.

Lastly, since Outerplane Splitting Number can be solved in polynomial time for maximal planar graphs but is hard for plane biconnected graphs, there is a complexity gap to be closed when faces of degree more than three are involved.

CHAPTER 5

# Bipartite Graphs

In this chapter, we focus on another restricted graph class by studying bipartite graphs. These graphs are very common for practical use, and have many properties that we can take advantage of within the vertex splitting problem. This chapter is based on a collaboration between Reyan Ahmed, Patrizio Angelini, Michael A. Bekos, Giuseppe Di Battista, Michael Kaufmann, Philipp Kindermann, Stephen Kobourov, Martin Nöllenburg, Antonios Symvonis and Markus Wallinger. Our main contribution to this project can be found in Section 5.3, but other sections have been kept for completeness and coherence. The full paper has been accepted to IEEE Computer Graphics and Application.

Multilayer networks are used in many applications to model complex relationships between different sets of entities in interdependent subsystems [MRA+21]. When analyzing and exploring the interaction between two such subsystems $S_t$ and $S_b$, bipartite or 2-layer networks arise naturally. The nodes of the two subsystem are modeled as a bipartite vertex set $V = V_t \cup V_b$ with $V_t \cap V_b = \emptyset$, where $V_t$ contains the vertices of the first subsystem $S_t$ and $V_b$ those of $S_b$. The inter-layer connections between $S_t$ and $S_b$ are modeled as an edge set $E \subseteq V_t \times V_b$, forming a bipartite graph $G = (V_t \cup V_b, E)$. Visualizing this bipartite graph $G$ in a clear and understandable way is a key requirement to designing tools for visual network analysis [PFH+18].

In a *2-layer graph drawing* of a bipartite graph the vertices are drawn as points on two distinct parallel lines $\ell_t$ and $\ell_b$, and edges are drawn as straight-line segments [EW94]. The vertices in $V_t$ (*top vertices*) lie on $\ell_t$ (the *top layer*) and those in $V_b$ (*bottom vertices*) lie on $\ell_b$ (the *bottom layer*). In addition to direct applications of 2-layer networks for modeling the relationships between two communities as mentioned above [PFH+18], such drawings also occur in tanglegram layouts for comparing phylogenetic trees [SZH11] or as components in layered drawings of directed graphs [STT81] and between consecutive axes in hive plots [KBJM12].

The primary optimization goal for 2-layer graph drawings is to find permutations of one or both vertex sets $V_t$, $V_b$ to minimize the number of edge crossings. While the existence of a crossing-free 2-layer drawing can be tested in linear time [EMW86], the crossing minimization problem is NP-complete even if the permutation of one layer is given [EW94]. Hence, both fixed-parameter algorithms [KT15] and approximation algorithms [DF01] have been studied. Further, graph layouts on two layers have also been investigated in the area of graph drawing beyond planarity [DLM19]. However, from a practical point of view, minimizing the number of crossings in 2-layer drawings may still result in visually complex drawings [JM97]. Hence, we focus here on using vertex splitting to construct readable 2-layer drawings.

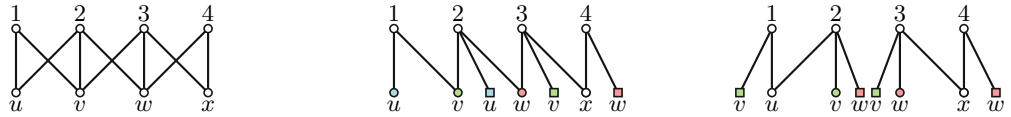
We study variations of the algorithmic problem of constructing planar or crossing-minimal 2-layer drawings with vertex splitting. When visualizing biological networks, for example graphs defined on anatomical structures and cell types in the human body [PBHI$^+$], the two vertex sets of $G$ play different roles. Thus, we consider the variation where only the vertices on one side of the layout may be split. While only the bottom vertices may be split, the top vertices may either be specified with a given context-dependent input ordering, e.g., alphabetically, following a hierarchy structure, or sorted according to an important measure, or allowed to arbitrarily permute them to perform fewer vertex splits.

**Structure of the chapter:** In Section 5.2, it is shown that for a given integer $k$ it is NP-complete to decide whether $G$ admits a planar 2-layer drawing with an arbitrary permutation on the top layer and at most $k$ vertex splits on the bottom layer (see Theorem 9). In Section 5.4 it is shown that NP-completeness also holds if instead of minimizing the number of splits, we minimize the number of vertices that are split (see Theorem 11). If, however, the vertex order of $V_t$ is given, then two linear-time algorithms to compute planar 2-layer drawings are presented, one minimizing the total number of splits (see Theorem 10), and one minimizing the number of split vertices (see Theorem 12). In Section 5.3, we study the setting in which the goal is to minimize the number of crossings (but not necessarily remove all of them) using a prescribed total number of splits. For this setting, we prove NP-completeness even if the vertex order of $V_t$ is given (see Theorem 13). On the other hand, we provide an XP-time algorithm parameterized by the number of allowed splits (see Theorem 14), which, in other words, means that the algorithm has a polynomial running time for any fixed number of allowed splits.

## 5.1 Preliminaries

We denote the order of the vertices in $V_t$ and $V_b$ in a 2-layer drawing by $\pi_t$ and $\pi_b$, respectively. If a vertex $u$ precedes a vertex $v$, then we denote it by $u \prec v$. Although 2-layer drawings are defined geometrically, their crossings are fully described by $\pi_t$ and $\pi_b$, as in the following folklore lemma.

**Lemma 17.** *Let $\Gamma$ be a 2-layer drawing of a bipartite graph $G = (V_t \cup V_b, E)$. Let $(v_1, u_1)$ and $(v_2, u_2)$ be two edges of $E$ such that $v_1 \prec v_2$ in $\pi_t$. Then, edges $(v_1, u_1)$ and $(v_2, u_2)$*




(a) (b) (c)

Figure 5.1: (a) Instance $G$. (b) A solution with three splits, involving three different vertices, that is optimal for CRS. (c) Optimal CRSV solution with two split vertices.

*cross each other in* $\Gamma$ *if and only if* $u_2 \prec u_1$ *in* $\pi_b$.

In the following we formally define the problems we study. For all of them, the input contains a bipartite graph $G = (V_t \cup V_b, E)$ and a split parameter $k$.

**Problem 7** (Crossing Removal with $k$ Splits – CRS($k$))**.** Given a bipartite graph $G$, can $G$ be transformed into a planar 2-layer drawing after applying at most $k$ vertex-splits to the vertices in $V_b$.

**Problem 8** (Crossing Removal with $k$ Split Vertices – CRSV($k$))**.** Given a bipartite graph $G$, can $G$ be transformed into a planar 2-layer drawing after splitting at most $k$ original vertices of $V_b$.

**Problem 9** (Crossing Minimization with $k$ Splits – CMS($k, M$))**.** Given a bipartite graph $G$, can $G$ be transformed into a planar 2-layer drawing with at most $M$ crossings after applying at most $k$ vertex-splits to the vertices in $V_b$, where $M$ is an additional integer specified as part of the input.

Note that in CRSV($k$), once we decide to split an original vertex, then we can further split its copies without incurring any additional cost. The example in Fig. 5.1 demonstrates the difference between the two problems concerning the removal of all crossings.

For all problems, we refer to the variant where the order $\pi_t$ of the vertices in $V_t$ is given as part of the input by adding the suffix *"with Fixed Order"*.

The following lemma implies conditions under which a vertex split must occur.

**Lemma 18.** *Let $G = (V_t \cup V_b, E)$ be a bipartite graph and let $u \in V_b$ be a bottom vertex adjacent to two top vertices $v_1, v_2 \in V_t$, with $v_1 \prec v_2$ in $\pi_t$. In any planar 2-layer drawing of $G$ in which $u$ is not split, we have that:*

1. *A top vertex that appears between $v_1$ and $v_2$ in $\pi_t$ can only be adjacent to $u$;*
2. *In $\pi_b$, $u$ is the last neighbor of $v_1$ and the first neighbor of $v_2$.*

*Proof.* If there is a top vertex $v'$ between $v_1$ and $v_2$ adjacent to a bottom vertex $u' \neq u$, then $(v', u')$ crosses $(v_1, u)$ or $(v_2, u)$. If there is a neighbor $u''$ of $v_1$ after $u$ in $\pi_b$, then the edges $(v_1, u'')$ and $(v_2, u)$ cross. A symmetric argument holds when there is a neighbor of $v_2$ before $u$ in $\pi_b$. □

## 5.2 Crossing Removal with Bounded Splits

In this section, we prove that the CRS($k$) problem is NP-complete in general and linear-time solvable when the order $\pi_t$ of the top vertices is part of the input.

**Theorem 9.** *The CRS($k$) problem is* NP*-complete.*

*Proof.* The problem belongs to NP since, given a set of at most $k$ splits for the vertices in $V_b$, we can check whether the resulting graph is planar 2-layer [EMW86].

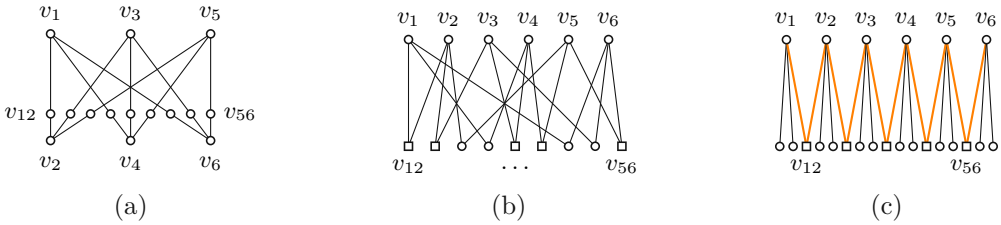
We use a reduction from the *Hamiltonian Path* problem to show the NP-hardness; see Fig. 5.2. Given an instance $G = (V, E)$ of the Hamiltonian Path problem, we denote by $G'$ the bipartite graph obtained by subdividing every edge of $G$ once. We construct an instance of the CRS($k$) problem by setting the top vertex set $V_t$ to consist of the original vertices of $G$, the bottom vertex set $V_b$ to consist of the subdivision vertices of $G'$, and the split parameter to $k = |E| - |V| + 1$. The reduction can be easily performed in linear time. We prove the equivalence.

Suppose that $G$ has a Hamiltonian path $v_1, \ldots, v_n$. Set $\pi_t = v_1, \ldots, v_n$, and split all the vertices of $V_b$, except for the subdivision vertex of the edge $(v_i, v_{i+1})$, for each $i = 1, \ldots, n-1$. This results in $|V_b| - (n-1)$ splits, which is equal to $k$, since $|V_b| = |E|$ and $n = |V|$. We then construct $\pi_b$ such that, for each $i = 1, \ldots, n-1$, all the neighbors of $v_i$ appear before all the neighbors of $v_{i+1}$, with their common neighbor being the last neighbor of $v_i$ and the first of $v_{i+1}$. This guarantees that both conditions of Lemma 18 are satisfied for every vertex of $V_b$. Together with Lemma 17, this guarantees that the 2-layer drawing is planar.

Suppose now that $G'$ admits a planar 2-layer drawing with at most $|E| - |V| + 1$ splits. Since $|E| = |V_b|$ and every vertex of $V_b$ has degree exactly 2 (subdivision vertices), there exist at least $|V| - 1$ vertices in $V_b$ that are not split. Consider any such vertex $u \in V_b$. By C.1 of Lemma 18, the two neighbors of $u$ are consecutive in $\pi_t$. Also, these vertices are connected in $G$ by the edge whose subdivision vertex is $u$. Since this holds for each of the at least $|V| - 1$ non-split vertices, we have that each of the $|V| - 1$ distinct pairs of consecutive vertices in $V_t$ (recall that $V_t = V$) is connected by an edge in $G$. Thus, $G$ has a Hamiltonian path. □

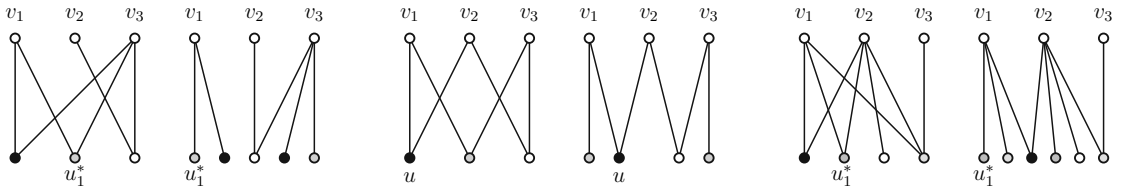
Next, we focus on the optimization version of the CRS($k$) with the Fixed Order problem. Our recursive algorithm considers a constrained version of the problem, where the first neighbor in $\pi_b$ of the first vertex in $\pi_t$ may be prescribed. At the outset of the recursion, there exists no prescribed first neighbor. The algorithm returns the split vertices in $V_b$ and the corresponding order $\pi_b$.

In the base case, there is only one top vertex $v$, i.e., $|V_t| = 1$. Since all vertices in $V_b$ have degree 1, no split takes place. We set $\pi_b$ to be any order of the vertices in $V_b$ where the first vertex is the prescribed first neighbor of $v$, if any.




(a) (b) (c)

Figure 5.2: Theorem 9. (a) Subdivided graph $G'$. (b) Instance of CRS($k$). (c) Splits are minimized if and only if $G$ has a Hamiltonian path.




(a) (b) (c)

Figure 5.3: Algorithm for CRS($k$) with Fixed Order optimization. Vertices in $N^+$ are colored in shades of gray. (a) Case 1, (b) Case 2, and (c) Case 3.

In the recursive case when $|V_t| > 1$, we label the vertices in $V_t$ as $v_1, \ldots, v_{|V_t|}$, according to $\pi_t$. If the first neighbor of $v_1$ is prescribed, we denote it by $u_1^*$. Also, we denote by $N^1$ the set of degree-1 neighbors of $v_1$, and by $N^+$ the other neighbors of $v_1$. Note that only the vertices in $N^+$ are candidates to be split for $v_1$. In particular, by C.1 of Lemma 18, a vertex in $N^+$ can avoid being split only if it is also incident to $v_2$. Further, since any vertex in $N^+$ that is not split must be the last neighbor of $v_1$ in $\pi_b$, by C.2 of Lemma 18, at most one of the common neighbors of $v_1$ and $v_2$ will not be split. Analogously, if $u_1^*$ is prescribed, then it must be split, unless $v_1$ has degree 1.

In view of these properties, we distinguish three cases based on the common neighborhood of $v_1$ and $v_2$. In all cases, we will recursively compute a solution for the instance composed of the graph $G' = (V'_t \cup V'_b, E')$ obtained by removing $v_1$ and the vertices in $N^1$ from $G$, and of the order $\pi'_t = v_2, \ldots, v_{|V_t|}$. We denote by $\pi'_b$ and $s'$ the computed order and the number of splits for the vertices in $V'_b$. In the following we specify for each case whether the first neighbor of $v_2$ in the new instance is prescribed or not, and how to incorporate the neighbors of $v_1$ into $\pi'_b$.

**Case 1: $v_1$ and $v_2$ have no common neighbor**; see Figure 5.3a. In this case, we do not prescribe the first neighbor of $v_2$ in the instance composed of $G'$ and $\pi'_t$. To compute a solution for the original instance, we split each vertex in $N^+$ so that one copy becomes incident only to $v_1$. We construct $\pi_b$ by selecting the prescribed vertex $u_1^*$, if any, followed by the remaining neighbors of $v_1$ in any order and, finally, by appending $\pi'_b$. This results in $s = |N^+| + s'$ splits.

**Case 2: $v_1$ and $v_2$ have exactly one common neighbor $u$.** If $u = u_1^*$ and $v_1$ have a

degree larger than 1, then $u$ cannot be the last neighbor of $v_1$ and must be split. Thus, we perform the same procedure as in Case 1. Otherwise, in the instance composed of $G'$ and $\pi'_t$, we set $u$ as the prescribed first neighbor of $v_2$; see Figure 5.3b. To compute a solution for the original instance, we split each vertex in $N^+$, except $u$, so that one copy becomes incident only to $v_1$. We construct $\pi_b$ by selecting the prescribed vertex $u_1^*$, if any, followed by the remaining neighbors of $v_1$ different from $u$ in any order and, finally, by appending $\pi'_b$. This results in $s = |N^+| - 1 + s'$ splits.

**Case 3: $v_1$ and $v_2$ have more than one common neighbor.** If $v_1$ and $v_2$ have exactly two common neighbors $u, u'$ and one of them is $u_1^*$, say $u = u_1^*$, then $u$ cannot be the last neighbor of $v_1$, as $v_1$ has degree larger than 1. Thus, we proceed exactly as in Case 2, using $u'$ as the only common neighbor of $v_1$ and $v_2$.

Otherwise, there are at least two neighbors of $v_1$ different from $u_1^*$; see Figure 5.3c. We want to choose one of these vertices as the last neighbor of $v_1$, so that it is not split. However, the choice is not arbitrary as this may affect the possibility for $v_2$ to save the split for a neighbor it shares with $v_3$. In the instance composed of $G'$ and $\pi'_t$, we do not prescribe the first vertex of $v_2$. To compute a solution for the original instance, we simply choose as the last neighbor of $v_1$ any of its common neighbors with $v_2$ that has not been set as the last neighbor of $v_2$ in $\pi'_b$. Such a vertex, say $u$, always exists since $v_1$ and $v_2$ have at least two common neighbors different from $u_1^*$, and can be moved to become the first vertex in $\pi'_b$. Specifically, we split all the vertices in $N^+$, except for $u$, so that one copy becomes incident only to $v_1$. We construct $\pi_b$ by selecting the prescribed vertex $u_1^*$, if any, followed by the remaining neighbors of $v_1$ different from $u$ in any order. We then modify $\pi'_b$ by moving $u$ to be the first vertex. Note that this operation does not affect planarity, as it only involves reordering the set of consecutive degree-1 vertices incident to $v_2$. Finally, we append the modified $\pi'_b$. This results in $s = |N^+| - 1 + s'$ splits.

**Theorem 10.** *For a bipartite graph $G = (V_t \cup V_b, E)$ and an order $\pi_t$ of $V_t$, the optimization version of CRS(k) with Fixed Order can be solved in $O(|E|)$ time.*

*Proof.* By construction, for each $i = 1, \ldots, |V_t|-1$, all neighbors of $v_i$ precede all neighbors of $v_{i+1}$ in $\pi_b$. Thus, by Lemma 17, the drawing is planar. The minimality of the number of splits follows from Lemma 18, as discussed before the case distinction. In particular, any minimum-splits solution can be shown to be equivalent to the one produced by our algorithm. The time complexity follows as each vertex only needs to check its neighbors a constant number of times. □

We conclude this section by mentioning that the CRS($k$) problem had already been considered, under a different terminology, in the context of molecular QCA circuits design [CCH+07]. Here, the problem was claimed to be NP-complete, without providing a formal proof. In the same work, when the order $\pi_t$ of the top vertices is part of the input, an alternative algorithm was proposed based on the construction of an auxiliary graph that has super-linear size. Exploiting linear-time sorting algorithms and observations that allow avoiding explicitly constructing all edges of this graph, the authors were able

to obtain a linear-time implementation. We believe that our algorithm of Theorem 10 is simpler and more intuitive, and directly leads to a linear-time implementation.

## 5.3 Crossing Removal with Bounded Split Vertices

In this section, we prove that the CRSV($k$) problem is NP-complete in general and linear-time solvable when the order $\pi_t$ of the top vertices is part of the input. Ahmed et al. [AKK22] showed that CRSV($k$) is FPT when parameterized by $k$. To prove the NP-completeness we can use the reduction of Theorem 9. In fact, in the graphs produced by that reduction all vertices in $V_b$ have degree 2. Hence, the number of vertices that are split coincides with the total number of splits.

**Theorem 11.** *The CRSV(k) problem is* NP*-complete.*

For the version with Fixed Order, we first use C.1 of Lemma 18 to identify vertices that need to be split at least once, and repeatedly split them until each has degree 1. For a vertex $u \in V_b$, we can decide if it needs to be split by checking whether its neighbors are consecutive in $\pi_t$ and, if $u$ has degree at least 3, all its neighbors different from the first and last have degree exactly 1.

We first perform all necessary splits. For each $i = 1, \dots, |V_t| - 1$, consider the two consecutive top vertices $v_i$ and $v_{i+1}$. If they have no common neighbor, no split is needed. If they have exactly one common neighbor $u$, then we set $u$ as the last neighbor of $v_i$ and the first of $v_{i+1}$, which allows us not to split $u$, according to C.2. Since $u$ did not participate in any necessary split, if $u$ is also adjacent to other vertices, then all its neighbors have degree 1, except possibly the first and last. Hence, C.2 can be guaranteed for all pairs of consecutive neighbors of $u$.

Otherwise, $v_i$ and $v_{i+1}$ have at least two common neighbors and thus have degree at least 2. Hence, all common neighbors of $v_i$ and $v_{i+1}$ must be split, except for at most one, namely the one that is set as the last neighbor of $v_i$ and as the first of $v_{i+1}$. Since all these vertices are incident only to $v_i$ and $v_{i+1}$, as otherwise they would have been split by C.1, we can arbitrarily choose any of them, without affecting the splits of other vertices.

At the end we construct the order $\pi_b$ so that, for each $i = 1, \dots, |V_t| - 1$, all the neighbors of $v_i$ precede all the neighbors of $v_{i+1}$, and the unique common neighbor of $v_i$ and $v_{i+1}$, if any, is the last neighbor of $v_i$ and the first of $v_{i+1}$. By Lemma 17, this guarantees planarity. Identifying and performing all unavoidable splits and computing $\pi_b$ can be easily done in $O(|E|)$ time. Since we only performed unavoidable splits, as dictated by Lemma 18, we have the following.

**Theorem 12.** *For a bipartite graph $G = (V_t \cup V_b, E)$ and an order $\pi_t$ of $V_t$, the optimization version of CRSV(k) with Fixed Order minimizing the number of split vertices can be solved in $O(|E|)$ time.*

## 5.4 Crossing Minimization with Bounded Splits

In this section we consider minimizing crossings, by applying at most $k$ splits. We first prove NP-completeness of the decision problem CMS$(k, M)$ with Fixed Order and then give a polynomial-time algorithm assuming the integer $k$ is a constant.

**Theorem 13.** *For a bipartite graph $G = (V_t \cup V_b, E)$, an order $\pi_t$ of $V_t$, and integers $k, M$, problem CMS$(k, M)$ with Fixed Order is* NP*-complete.*

*Proof.* We reduce from the NP-complete Decision Crossing Problem (DCP) [EW94], where given a bipartite 2-layer graph with one vertex order fixed, the goal is to find an order of the other set such that the number of crossings is at most a given integer $M$. Given an instance of DCP, i.e.,a 2-layer graph $G = (V_t \cup V_b, E)$, with ordering $\pi_t$ of $V_t$ and integer $M$, we construct an instance $G'$ of CMS$(k, M)$ where $k = |V_b|$. First let $G' = (V_t' \cup V_b', E')$ be a copy of $G$. We give an arbitrary ordering $\pi_b$ to the vertices of $V_b'$. We then add, respectively, to each vertex set $V_t$ and $V_b$ a set $U_t$ and $U_b$ of $M + 1$ vertices and connect each $u \in U_t$ to exactly one $v \in U_b$, forming a matching of size $M + 1$. We add the vertices of $U_t$ to $\pi_t$ (resp. $U_b$ to $\pi_b$) after all the vertices of $V_t$ ($V_b$). We lastly add a set $W_t$ of $k$ vertices to $V_t'$, placed at the end of $\pi_t$, such that each $w_i \in W_t$ $(i = 1, \ldots, k)$ has exactly one neighbor $v_i \in V_b$ and vice versa (see Fig. 5.4a).

Given an ordering $\pi_b^*$ of $V_b$ that results in a drawing of $G$ with at most $M$ crossings, we show that we can solve the CMS$(k, M)$ instance $G'$. In $G'$, we split each vertex of $V_b$ to obtain the sets $V_b^1$ and $V_b^2$ in which we place exactly one copy of each original vertex. We place $V_b^2$ after the vertices of $U_b$ in $\pi_b$ in the same order that the vertices of $W_t$ appear in $\pi_t$ and draw a single edge between the copies and their neighbor in $W_t$. We place $V_b^1$ before the vertices of $M_2$ in $\pi_b$ in the same order as in $\pi_b^*$. The graph induced by $V_b^1$ and $V_t$ is the same graph as $G$, hence it has at most $M$ crossings. Since $V_t$ only has neighbors in $V_b$ and all those neighbors are in $V_b^1$, it has no other outgoing edges, similarly, all edges incident to vertices in $W_t$ are assigned to the copies in $V_b^2$. The remaining graph is crossing-free as the vertices in $U_t$ and $W_t$ form a crossing-free matching with the vertices in $U_b$ and $V_b^2$.

Conversely, let $G^*$ be a 2-layer drawing obtained from $G'$ after $k$ split operations that has at most $M$ crossings. Since each vertex $v \in V_b$ has a neighbor $w \in W_t$, it induces $M + 1$ crossings with edges induced by the vertices in $U_t \cup U_b$. Since the vertices in $U_b$ have a single neighbor, they cannot be split, thus every vertex in $V_b$ is split once, and their neighborhood are partitioned for each copy in the following way: one copy receives the neighbor in $W_t$ and one copy receives the remaining neighbors, which are in $V_t$ (see Fig. 5.4b), thus avoiding the at least $M + 1$ crossings induced by $U_t \cup U_b$. Any other split would imply at least $M + 1$ crossings. The graph induced by the copies that receive the neighbors in $V_t$ has at most $M$ crossings, thus, the ordering found for those copies is a solution to the DCP instance $G$. □

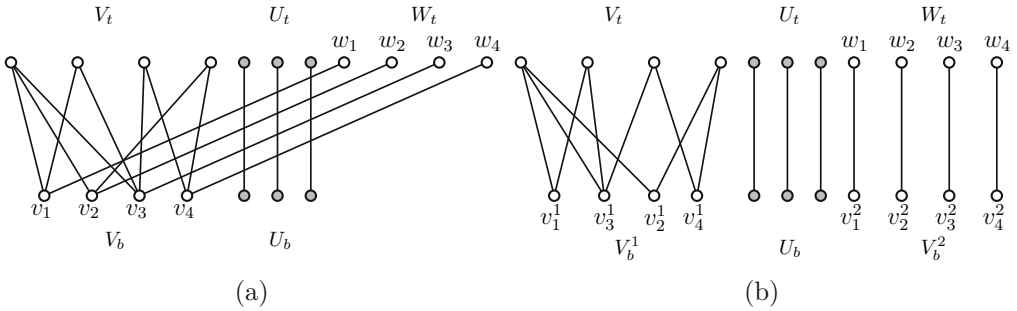



Figure 5.4: Instance of CMS($k$) constructed from a DCP instance, in light gray the vertices in $U_t \cup U_b$ (a) before the splitting operation, (b) after splitting

Next, we present a simple XP-time algorithm for the crossing minimization version of CMS($k, M$), parameterized by the number $k$ of splits, i.e., the algorithm runs in polynomial time $O(n^{f(k)})$, where $n$ is the input size, $k$ is the parameter, and $f$ is a computable function. Let $G = (V_t \cup V_b, E)$ be a 2-layer graph with vertex orders $\pi_t$ and $\pi_b$ and let $k$ be the desired number of splits. Our algorithm executes the following steps. First, it determines a set of splits by choosing $k$ times a vertex from the $n$ vertices in $V_b$ – we enumerate all options. For any vertex $v \in V_b$ split $i$ times in the first step, $v$ is replaced by the set of copies $\{v_1, ..., v_{i+1}\}$. The neighborhood $N(v)$ of a vertex $v \in V_b$ is a subset of $V_t$ ordered by $\pi_t$. We partition this ordered neighborhood into $i+1$ consecutive subsets, i.e., for each subset, all its elements are sequential in $N(v)$ – again, we enumerate all possible partitions. Each set is assigned to be the neighborhood of one of the copies of $v$. The algorithm then chooses an ordering of all copies of all split vertices and attempts all their possible placements by merging them into the order $\pi_b$ of the unsplit vertices of $V_b$. The crossing number of every resulting layout is computed and the graph with minimum crossing number yields the solution to our input. It remains to show that the running time of this algorithm is polynomial for constant $k$.

**Theorem 14.** *For a 2-layer graph $G = (V_t \cup V_b, E)$ with vertex orders $\pi_t, \pi_b$ and a constant $k \in \mathbb{N}$ we can minimize the number of crossings by applying at most $k$ splits in time $O(n^{4k})$.*

*Proof.* Let $G^*$ be a crossing-minimal solution after applying $k$ splits on $V_b$ and let us assume that our algorithm would not find a solution with this number of crossings. As our algorithm considers all possibilities to apply $k$ splits, it also attempts the splits applied in $G^*$. Similarly, the neighborhood partition of $G^*$ and the copy placement are explicitly considered by the algorithm as it enumerates all possibilities. Hence a solution at least as good as $G^*$ is found, proving correctness.

Let $n_t = |V_t|$ and $n_b = |V_b|$ with $n = n_t + n_b$. The algorithm initially chooses $k$ times from $n_b$ vertices leading to $n_b^k$ possible sets of copies. Since a vertex has degree at most $n_t$, there are at most $n_t^k$ possible neighborhoods for each copy. Additionally, there are $(2k)!$ orderings of at most $2k$ copies. Lastly, there are $n_b^{2k}$ possible placement of the $2k$

ordered copies between the at most $n_b$ unsplit vertices in $\pi_b$. This leads to an overall runtime of $O((2k)! \cdot n^{4k}) = O(n^{4k})$ to iterate through all possible solutions and select the one with a minimum number of crossings. □

## 5.5 Chapter Conclusion

In this chapter, we considered the vertex splitting applied to bipartite graphs. We considered both abstract graphs, and layered drawings with a given vertex ordering. We found that while most crossing minimization or removal problems were hard, when given an ordering of the non-split vertex layer, one could in linear time compute a planar drawing of the graph either minimizing the number of splits, or minimizing the number of split vertices.

One could consider the case o, which vertices might be split in both layers, and partial ordering given. In general, layered bipartite drawings have a lot of structural properties, thus, they would make ideal candidates to study vertex splitting to improve other quality measures. For example, when visualizing large bipartite graphs, a natural goal is to arrange the vertices so that a small window can capture all the neighbors of a given node, i.e., minimize the maximum distance between the first and last neighbors of a top vertex in the order of the bottom vertices.

Lastly, while we found that the problem can be solved in linear time for layered drawing, one could investigate if the problem is still polynomial time solvable when considering general bipartite graph drawings.

CHAPTER 6

# Practical Considerations

The main motivation for studying vertex splitting, is its application to visualizing non-planar graphs. In previous chapters, we focused on theoretical approaches to the problem. While we found that vertex splitting for crossing minimization was largely a hard problem, we were able to gain significant insight into the problem's main challenges. Additionally, we were also able to introduce some efficient algorithms for restricted subproblems. In this chapter, our goal is to extend this knowledge towards efficient solutions for vertex splitting in graph drawings. Specifically, we now focus on practical algorithms for our main problem of crossing minimization. We introduce an algorithmic pipeline, where user input can be incorporated at each stage. By doing so, we aim to outline the challenges and design goals for practical vertex splitting solutions.

One of the primary challenges of graph visualizations is to effectively communicate information contained in large and dense graphs. Many of the data sets we are interested in are often too big to be efficiently displayed, which results in messy "hairball"-like layouts. When visualizing a graph, we must be able to distinguish the different entities, and to also visualize and analyze their relationships. To create representations for such large networks, our algorithms are often limited, and we must rely on heuristic methods to generate layouts. Force directed graph drawing methods are common for such a use case, but they offer no guarantee of readability or aesthetic qualities.

The challenge of visualizing large graphs has led to the development of various techniques aimed at improving graph drawings. One such method is graph summarization [LSDK18], where the task is to create a simplified version of an input graph by grouping nodes or edges based on some criteria, such as their similarity or such as structural properties like an underlying hierarchy. Summarization techniques include node aggregation [EF10], where nodes are grouped together based on shared attributes or similarly, edge aggregation methods like edge bundling. With edge bundling, edges sharing a similar orientation are clustered [ZXYQ13], thus densifying dense regions, and rendering the rest of the drawing more sparse. One can also consider graph simplification methods, like vertex

deletion [JdKW21]. While it is a powerful tool to make a drawing more legible, deletion in general is not an ideal solution as the loss of information is too detrimental to the faithfulness of the data being presented. Vertex splitting shares many similarities with vertex deletion, and it offers an interesting alternative to deletion as it permits us to retain all of the information at the cost of increasing the visual load through the addition of extra vertices.

While we focused on crossing removal in Chapter 3 and 5, vertex splitting can also be used to improve other graph aesthetics, or to target other graph properties as shown in Chapter 4. Similarly, splitting has been used in visualization to achieve different goals, for example to transform a network into a hierarchical one [LPP+06], or to visualize a network as a ontology [ERG02]. It has also been used in social networks to display interlinked group as matrices by duplicating actors that are connected to multiple groups [HBF08]. Splitting has also been used to visualize Euler diagrams [HRD10] or biological networks [WNSV19]. In the case of biological network, as they are often very large and complex but their representations are important for researchers, vertex duplication is often used to obtain layouts with a small number of crossings. Currently, metabolic pathway graphs [KFS+22] are largely manually drawn [NDG+17]. As metabolic pathways often have some vertices of very high degree (for example the water molecule is involved in a large number of chemical reactions), vertex splitting or deletion is required to obtain a legible drawing. Nielsen et al. [NOM+19] used machine learning to automate the creation of these graphs, and guide the vertex splitting operation, but as the training data is limited to already existing manually drawn graphs, this method cannot be easily extended to more general instances. While vertex splitting is a very practical and powerful tool in graph visualization, it has not been studied closely, and is limited to heuristics and heavily targeted applications. Eades and Mendonça [EdMN95] studied the operation in the setting of tension minimization, but to our knowledge, vertex splitting has not yet been studied in other general settings.

In this chapter, we focus on the crossing minimization problem, as it is the most thoroughly studied from a theoretical point of view. We initiate the discussion of efficient vertex splitting algorithms for graph visualization, and identify the challenges and benefits of the approach. We propose a series of methods to reduce crossings in graph drawings using vertex splitting through an algorithmic pipeline. As each of the steps of the pipeline can easily be guided by user preference, the algorithms we introduce are capable of taking human selection into account. While our algorithms attempt to compute optimal solutions to each subtasks, these solutions could instead be used as a set of hints to guide a user in their layout design. Lastly we evaluate the performance of the algorithms by developing a set of metrics targeted at crossing reduction via vertex splitting.

**Structure of the chapter:** We start by introducing in Section 6.1 the different steps of the pipeline, as well as the two main problems that were targeted by our implementation. We then focus on the first problem, computing a plane drawing through iterative vertex splitting, and cover in Section 6.2 the different algorithms that were used for this setting.

In Section 6.3, we consider a relaxation of the previous problem, through three algorithms that are targeted towards crossing minimization. We present an evaluation of our pipeline in Section 6.4, and the different algorithms we introduced. We first focus on a quantitative analysis of the algorithms' performance, and later present a case study on a smaller data set. Lastly, in Section 6.5 we cover the limitations of our implementation, as well as the remaining important questions that need to be further studied.

## 6.1 Preliminaries

In this chapter, we attempt to solve the Embedded Splitting Number problem (ESN) introduced in Chapter 3. Namely, given an input drawing $\Gamma$, to compute a plane drawing $\Gamma^*$ by applying splitting operations to the vertices of $\Gamma$. Additionally, we consider the Embedded Splitting Crossing Minimization problem (ESCM), a relaxation of ESN, where the output drawing $\Gamma^*$ must only contain fewer crossings than $\Gamma$.

To solve these problems, we consider a sequence of sub-tasks that follow the sequence established in Chapter 3. We first select a subset of vertices that will be split, and remove them from the drawing. Secondly, we split the selected vertices, and compute a neighborhood and embedding for the copies we obtained. As we are also considering crossing minimization, we must pay additional attention to the re-embedding tasks, specifically to the task of routing edges between copies and their neighbors. Lastly, as our focus is to propose a more manageable restricted system, we only attempt here to split a single vertex at a time. We detail in the following section each step of the pipeline.

### 6.1.1 A Three Step Pipeline

In Chapter 1, we highlighted the five different subproblems that must be solved during a splitting operation: (i) choosing a set of vertices to be split, (ii) choosing how many times each vertex should be split, (iii) choosing a partitioning of the original vertex's neighborhood, (iv) choosing an embedding for the new vertex, (v) routing the edges between the copies and its neighborhood partition. In this setting we have grouped them into the following three subproblems, illustrated in Fig. 6.1.

1. Choosing a set of vertices to be split (i)
2. Partitioning the original vertex's neighborhood (ii)-(iii)
3. Embedding the copy and connecting it to its neighbors (iv)-(v)

We now detail the challenges presented by each sub-task.

**Selecting a set of vertices to be split** Consider the input drawing $\Gamma$. Our task is to compute a set $S_\wedge$ of vertices to be split. To solve ESN, we must, at this step, ensure that the vertex set we compute is a solution to Embedded Vertex Deletion. For the

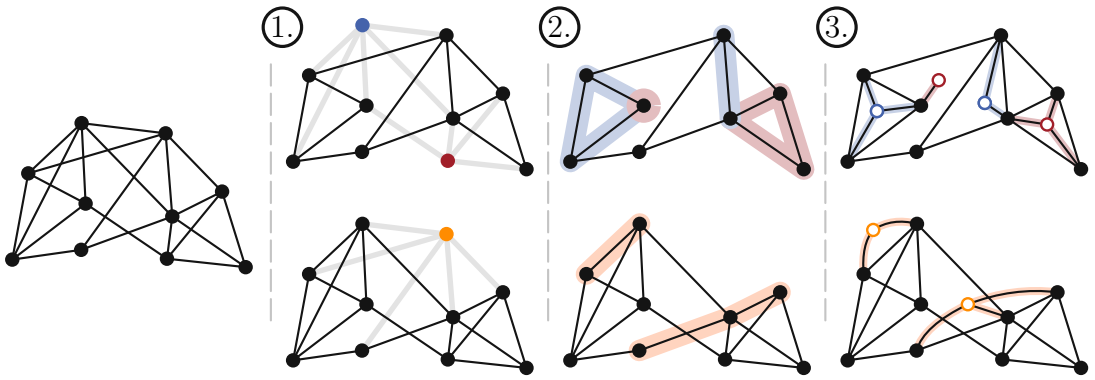



Figure 6.1: The three step pipeline for the input drawing on the left, on the top, an instance of ESN and an instance of ESCM on the bottom. In the first step, the colored disks represent the vertices that are selected te be split. In the second step, each continuous highlighted line corresponds to one partition of the removed vertex's neighborhood. In the third step we place the copies and connect them to their partition, in this example, we use straight lines for the top drawing and curves for the bottom one.

ESCM problem, it is sufficient if the drawing $\Gamma[V \setminus S_\wedge]$ has fewer crossings than $\Gamma$, thus, we focus on deleting vertices whose incident edges have a high number of crossings.

**Partitioning the Split Vertex's Neighborhood** Given a drawing $\Gamma'$ and a vertex $v \in S_\wedge$, the task is to partition the neighbors of $v$ such that each vertex in $N(v)$ is connected to a copy of $v$, and vertices in the same partition subset are connected to the same copy. Depending on the chosen algorithm for this task, the number of copies for each vertex is either manually set (or chosen by a user), or it is imposed by the algorithm. For example, when solving ESN, if $v$ is incident to three vertices $v_1, v_2, v_3$ such that no face in $\Gamma$ is incident to $v_1, v_2$ and $v_3$, then it must be split at least twice to ensure no crossing will be induced by connecting copies of $v$ their neighbors.

**Embedding and Connecting the Copies** The input at this step is a vertex $v \in S_\wedge$, a partition of $N(v)$ and a drawing $\Gamma'$. The task is to select a set of faces of $\Gamma'$ to embed the copies of $v$, and for a subset in $N(v)$, to connect its vertices to their corresponding copy. Depending on the chosen embedding method, we connect copies to their neighbors using straight-line edges or using curves. Curves can be required to solve ESN. For example, if a copy is embedded in a non-convex face or the outer face, then a straight-line edge is not necessarily crossing free[1].

Step two and three are done iteratively, meaning that one vertex is removed from $S_\wedge$ at a time and then split. Its copies are then embedded into $\Gamma$, and this operation is repeated until $S_\wedge$ is empty. This means that if there exists an edge $(u, v) \in E$, such that $u \in S_\wedge$

[1]Minimizing crossings in straight-line drawings while requiring the output to retain the straight-line property has been studied by Ehlers, Villedieu, Raidou and Wu (unpublished manuscript).

and $v \in S_\wedge$, and $u$ is split first, we are unable to connect a copy of $u$ to $v$ as $v$ has been removed from the drawing. Thus, only when we are splitting vertex $v$, can we select one of the copies of $u$ previously embedded in the drawing to be connected to a copy of $v$.

### 6.1.2 Metrics

As vertex splitting algorithms have, to our knowledge, not yet been closely studied or evaluated, we will propose here a novel set of natural metrics to gauge our algorithms.

- Our main task is to resolve edge crossings, thus the most important metric to consider is the ratio of the number of remaining edge crossing over the initial number of edge crossings present in the drawing, the *crossing resolution ratio*.
- Additionally, since each split introduces one vertex to the drawing (more accurately two new vertices are added and one is removed), we want to count how many splitting operations were performed, and how many input vertices were split at least once. Thus, we consider the *number of splits* that were executed.
- To understand how much our drawing is changed by the algorithm, we compute the *vertex splitting ratio*, the ratio of the number of vertices split over the total number of vertices in the input drawing.
- When routing curves through the outer face or non-convex faces, we might obtain very long and complex edges, thus we measure the *edge complexity* of the drawing by counting how many control points are required to route the curves between copies and their neighbors.
- Lastly, we will consider the *runtime* of our algorithm, as fast algorithms are paramount to an interactive setting.

## 6.2 Embedded Splitting Number

We begin by presenting the algorithms used to solve the Embedded Splitting Number problem. We have shown in Chapter 3 that ESN, as well as its subproblems are NP-complete, thus we make use of integer linear programming to compute solutions for each subtask.

### 6.2.1 Largest Planar Subdrawing

Given a drawing $\Gamma$, the task is to compute a plane drawing $\Gamma'$ by removing the smallest number of vertices from $\Gamma$. This means that for each crossing in $\Gamma$ at least one of the four involved vertices has to be removed from the graph. We call $S_H$ the set of removed vertices. If we model each crossing as a set of size four containing each vertex involved in the crossing, then the task is to find the smallest cardinality set $S_H$ such that each crossing set has at least one of its four endpoints in $S_H$.

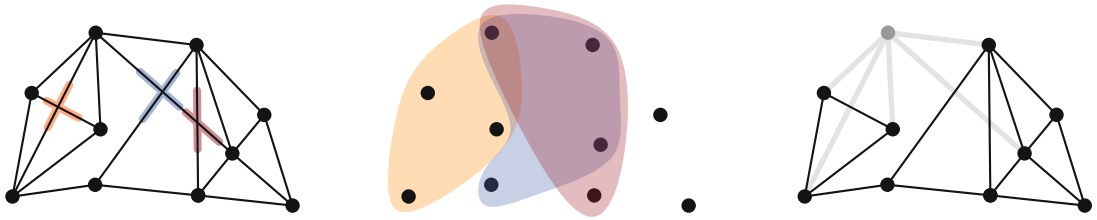


Figure 6.2: We solve the hitting set instance in the middle to find a set of vertices to split

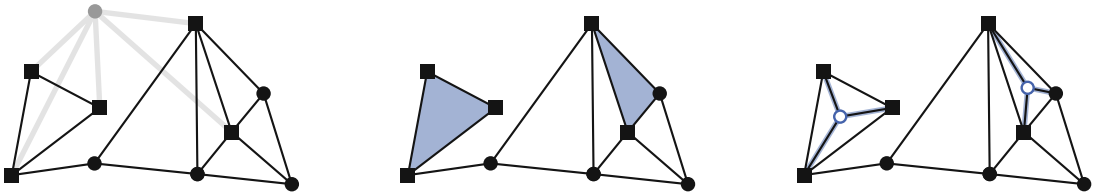


Figure 6.3: We solve the face cover instance in the middle to find the faces in which to embed our copies.

We find that this problem corresponds to a restricted version of the Hitting Set problem. In the hitting set problem, given a universe of elements, and a set $S$ of element subsets, the task is to compute a minimum cardinality subset of elements $S_e$ such that each subset in $S$ has an element in $S_e$, thus we model this problem using hitting set, as shown in Fig. 6.2 For a drawing $\Gamma$ of graph $G = (V, E)$, we first use a sweep-line algorithm to detect the crossings in $\Gamma$. Then, for each crossing, we create a set containing the subset of vertices that induces the crossing. We then compute $S_H$ using the following straightforward ILP formulation. For each vertex $v \in V$ in the input drawing, we create a variable $x_v$, such that $x_v = 1$ if vertex $v$ is part of $S_H$, and 0 otherwise. The objective function is the following.

$$\min \sum_{v} x_v \qquad \text{for } v \in V$$

Then, for a subset $S_c$ corresponding to a crossing between $(a, b)$ and $(c, d)$, we consider the following constraint which forces at least one of the vertices to be selected.

$$\sum x_a + x_b + x_c + x_d >= 1$$

We create such a constraint for every crossing in $\Gamma$. We call this vertex selection variant `v-HS`.

### 6.2.2 Face Cover

We showed in Chapter 3 that the Face Cover problem [BM88] can naturally be transformed into a vertex splitting problem. Given a graph $G$, a set of vertices $S$ and an integer $k$, the task is to find a set of at most $k$ faces such that each vertex in $S$ is incident to at least one of the $k$ selected faces. Consider now a vertex $v$ with neighborhood $N(v)$, and the face cover $S_F$ of $N(v)$ in $\Gamma$. By embedding a copy in each of the faces in $S_F$, we ensure that each vertex in $N(v)$ can be connected to a copy in $\Gamma$ without inducing any crossing, as it is necessarily incident to a face in $S_F$, as shown in Fig. 6.3.

The previously stated definition of face cover is insufficient when multiple vertices are being split. Consider two vertices $u$ and $v$, $u$ adjacent to $v$ that must be split. Consider $v$ has already been split and its copies $\{v_1, \ldots, v_i\}$ are in $\Gamma$. When splitting a vertex $u$, only one copy of $v$ must be incident to a face of the solution of the face cover instance. We now detail the ILP formulation to solve this modified face cover instance.

Given a drawing $\Gamma$ of graph $G = (V, E)$, we first compute the cyclic ordering of each vertex's neighborhood to obtain the set of faces $\mathcal{F}$ of $\Gamma$. Consider that $|\mathcal{F}| = n_F$, and the split vertex $u$ has neighborhood $N(u) = \{u_1, \ldots, u_m\}$.

The objective function is to minimize the number of faces that are part of the solution. A face $F_i$ is part of the solution if the variable $f_i$ has value 1.

$$\min \sum_i f_i \qquad \text{for } i \leq n_F$$

We use the variable $c_{i,j}$ to say that face $F_i$ *covers* a vertex $u_j \in N(u)$, meaning that the vertex $u_j$ is incident to face $F_i$, and face $F_i$ is part of the solution, thus we can connect $u_j$ to a copy placed in $F_i$. Additionally, we have to ensure that $u_j$ it covered by at least one face in $\mathcal{F}$ that is selected. Consider the subset $\mathcal{F}_j$ of $\mathcal{F}$ that corresponds to faces incident to $u_j$.

$$\sum_i c_{i,j} = 1 \qquad \text{for } i \text{ s.t. } F_i \in \mathcal{F}_j$$

This ensures that each neighbor of the copy is covered by a face. For simplicity, we model this as an equality to avoid having one neighbor covered by multiple faces. If the neighbor $v$ of $u$ has been split into copies $v_1, \ldots, v_\ell$, we use the variable $s_{i,j,k}$ to say that face $F_i$ covers the $k$th copy of neighbor $v$. This leads us to the following constraint.

$$\sum_{i,k} s_{i,j,k} = 1 \qquad \text{for } i \text{ s.t. } F_i \in \mathcal{F}_j \text{ and } k \text{ s.t. } v_k \text{ is a copy of } v$$

Similarly, this ensures that for one vertex $v$ that was split, exactly one of its copies is covered by a face. Note that enforcing neighbors of $u$ to be covered by exactly one face

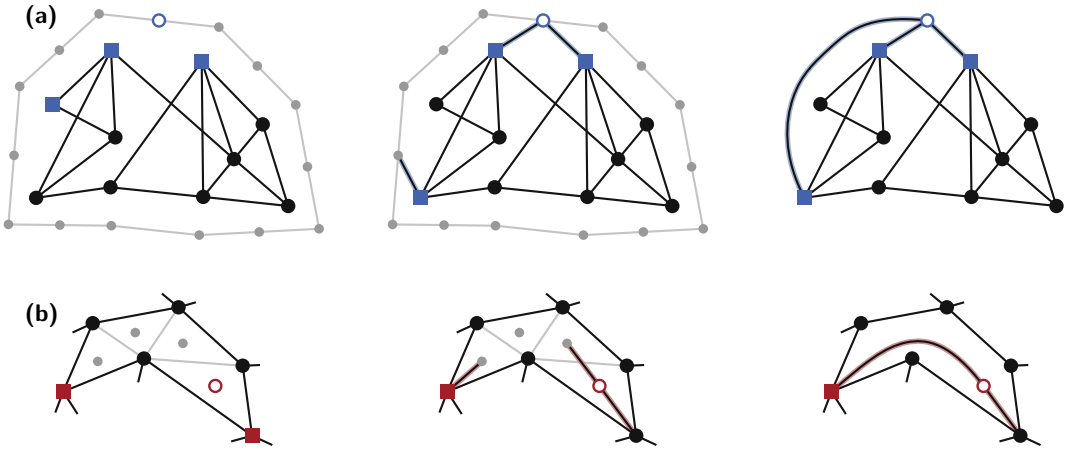



Figure 6.4: Examples of edges routed when a straight-line path induces crossings, **(a)** shows routing through the outer face, and **(a)** shows a routing example through an inner non convex face. The colored squares represent the neighbors that must be connected to the copy drawn as a circle

does not prevent them from being incident to several selected faces. Since neighbors can only be covered by faces selected in the solution, we set the following constraints.

$$\sum_i c_{i,j} \leq f_i \qquad \text{for } i \text{ s.t. } F_i \in \mathcal{F}_j$$

$$\sum_i s_{i,j,k} \leq f_i \qquad \text{for } i \text{ s.t. } F_i \in \mathcal{F}_j \text{ and } k \text{ s.t. } v_k \text{ is a copy of } v$$

This ensures that a face can only cover a neighbor of $u$, if it has value 1. We can then output the set of faces selected by the solver, as well as the set of vertices each of these selected faces cover. This vertex splitting algorithm variant is called `s-FC`.

### 6.2.3 Planar Edge Routing

Given a drawing $\Gamma$, a vertex $v$ to be split, a set of faces $S_F$, and for each face $F \in S_F$ a subset of $N(v)$ incident to $F$, the task is to find a crossing free drawing $\Gamma^*$ by embedding a copy of $v$ in each face of $S_F$ and connecting it to the corresponding subset of $N(v)$ with crossing free edges.

First we must compute coordinates inside a face $F$ to embed a copy $v_i$ in $F$. While the barycenter of the vertices incident to $F$ is a natural candidate position to embed the copy, the face might not be convex. Thus, we instead loop over any subset of three vertices incident to $F$, and check if the barycenter of that triangle lies inside of $F$. Once such a point is found, we use its coordinate as the coordinates of the copy.

As this is a discrete set of valid coordinates, it can lead to several vertices sharing the same embedding. Thus, we introduce a *wiggle*, a randomly generated small value, to shift the vertex slightly off of the position computed for it, to ensure no two vertices share the same coordinates. As this might induce a crossing, we generate new values for the wiggle until we can find a suitable position which does not induce additional crossings.

If the straight line edge between the copy of $v$ and its neighbor in $N(v)$ does not induce a crossing, then it is drawn. Otherwise, we must distinguish two scenarios.

- If the face is **non-convex**, we compute the Delaunay triangulation of its incident vertices. We place a control point at the barycenter of each triangle and connect two control points if their corresponding triangles share an edge. Then, we connect both the copy and its neighbor to the closest control point possible, and then find the shortest path through the remaining control points to finish connecting the two vertices (see Fig. 6.4(a)). This allows us to draw curves through non-convex faces to connect copies to their neighbors without inducing crossings.

- If the curve must be drawn through the **outer face**, we first compute the convex hull of the drawing. We then scale it up with a ratio of 1.2, and add several control points along this new large polygon. Then, as in the non-convex face step, we connect copies and neighbors to the control points, and then connect the control points together to obtain a curved crossing free edge (see Fig. 6.4(b)). This allows us to route curves around the drawing.

Using these algorithms, we can compute a plane drawing $\Gamma^*$. Note that while we make use of solvers and exact solutions, efficient algorithms exist for both hitting set and set cover [HL05, AP20] to compute approximation or small solutions.

## 6.3 Embedded Splitting Crossing Minimization

In this section we focus on the crossing removal through vertex splitting problem. To obtain a planar drawing when the input graph is dense, it is often necessary to split a very large number of vertices. This introduces a high amount of complexity to the graph by not only adding many copies, but additionally disturbing the input drawing significantly as well as disconnecting a large number of paths. While crossings are often detrimental in graph visualization, drawings with a small number of crossings can still be read efficiently [DLM19].

The algorithms introduced in Section 6.2 can also be used to solve ESCM, with the difference that a planarization of $\Gamma$ must be computed, as the drawing must be plane to compute its faces to solve face cover. The planarization is obtained by replacing edge crossings with graph vertices and subdividing intersecting edges into paths through the planarization vertices.

### 6.3.1 Vertex Selection

Our goal here is to remove the vertices responsible for the largest number of crossings in the drawing. To do this, we compute the number of crossings occurring on each edge, and for each vertex, sum the total number of crossings that are induced by all of its incident edges.

While we only focus on crossings to limit the scope of our investigation, one could also easily consider other quality metrics to select vertices, such as selecting the highest degree vertex, the vertex with the longest incident edges, a vertex that induces crossings with undesirable angles (desirable crossing angle would typically be right angles) and so on, depending on the aesthetic criteria preferred.

The vertices to be split can also be chosen by the user, and the previously computed metrics can provide guidance for this selection using the weights that we compute for the single vertex selection step. We call this metric based vertex selection method `v-M`.

### 6.3.2 Neighborhood Assignment

In this section we focus on the core task of the vertex splitting crossing minimization problem, partitioning the split vertices' neighborhoods. We propose three algorithms. One algorithm is the algorithm introduced in Section 3.2.2, that optimally minimizes crossings when splitting a single vertex $k$ times. The other two algorithms are based on graph community detection and on point clustering techniques.

**Embedding Copies Using the Dual Graph** We introduced in Chapter 3 an exact algorithm to split a single vertex $k$ times that optimally minimizes the number of crossings induced by re-embedding the copies into $\Gamma$. An example is shown in Fig. 6.5.

We begin by computing the planarization $\Gamma_p$ of $\Gamma$, (Fig. 6.5(a)), to compute the dual $D$ of $\Gamma_p$ (Fig. 6.5(b)). To find $D$, we first find the set of faces of $\Gamma_p$ and then create the dual by creating one vertex per identified face, and connecting any vertices that corresponds to faces are adjacent.

We can then compute the crossing minimal split of $v$ as described in Section 3.2.2. We loop over all subset of $k$ faces of $\Gamma_p$, and for such a subset $S_c$, we compute the shortest path in the dual between $v_i \in N(v)$ (more precisely, one of the faces incident $v_i$) and one of the faces in $S_c$ (Fig. 6.5(c)). Once every neighbor has been connected to a face in $S_c$, the lengths of each paths in the dual correspond to the minimum number of crossings induced by connecting all neighbors to copies embedded in this subset of faces. We then find the subset $S_c$ that induces the least number of crossings and return the corresponding embedding of the copies of $v$. We additionally output the ordered set of faces traversed to connect the neighbors of $v$ to their neighboring copy.

This algorithm has a running time of $O((|F| + |\mathcal{E}|) \cdot |N(v)| \cdot |F|^k)$. When considering a dense graph, the planarization can have a very large number of faces, meaning that in

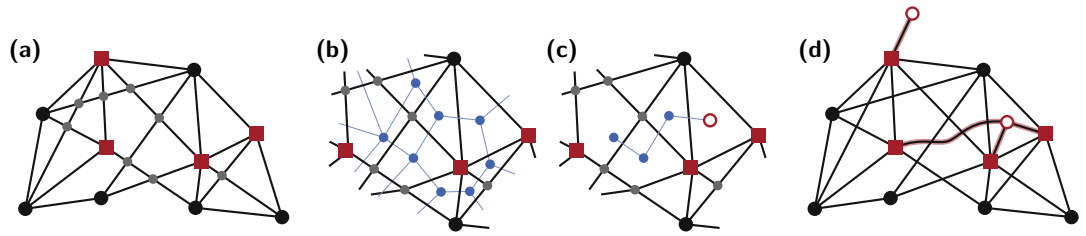



Figure 6.5: The red vertices are the neighbors of the split vertex. To compute the embedding that induces the minimal number of crossings, **(a)** we first compute the planarization of the graph (the small vertices have been introduced where crossings previously occurred), **(b)** which allows us to compute its dual (in blue), **(c)** we find the shortest path in the dual between the face where a copy could be embedded, and a face incident to a neighbor of the split vertex. Drawing **(d)** shows the embedding that would be computed if the solution in **(c)** is selected.

practice this algorithm performs quite poorly. Thus, when using this method, we choose to only split each vertex once, at the cost of having to split that same vertex again.

When the split vertex $v$ has a neighbor $u$ that has already been split, we cannot select for $u$ the closest copy of $v$ in the dual graph. Instead, for each copy of $u$ we find the closest face in which a copy of $v$ is embedded, and only add the length of the shortest path between one copy of $u$ and one copy of $v$ to the total number of crossings generated by that face subset. This algorithm is referred to as `s-D`.

**Community Detection-based Splitting** This well studied class of algorithms are used to identify groups of strongly connected vertices in a graph. In a complex network, nodes that are strongly inter-connected and share less connections to the rest of the graph can be identified as a community. If a vertex adjacent to multiple communities is split, it is then natural to have one copy representing that vertex for each community it was connected to.

To compute this, we use the Louvain algorithm [MFFP11], and obtain a *dendrogram* of the communities in the input graph. The dendrogram is a tree where each level of the tree corresponds to a partition of the nodes of our graph into communities. As in practice we find that the algorithm computes a larger number of smaller communities, we select the partition that has the lowest number of communities, to avoid creating too many copies of our split vertex. We then remove any partition that does not contain a neighbor of the split vertex. In the remaining communities, we remove any vertex that is not a neighbor of the split vertex, and return this vertex partition as an output. This algorithm is called `s-L`.

**Euclidean Clustering-based algorithm** Lastly, we make use of clustering methods. As our input is a drawing, every node has a given location. We abstract the graph structure and instead focus only on the coordinates of the neighbors of the split vertex. We then group these neighbors using Euclidean distance-based clustering with the k-

means algorithm [HW+79]. This allows us to choose how many copies of the split vertex we desire, by targeting a certain number of clusters in the solution. If two vertices are strongly connected but drawn far away from one another in the graph, connecting copies to each of these vertices might induce a large number of crossings, but it can also result in long spanning edges that increase the visual load of the drawing. Thus, this method results in splits whose impact is limited to a small area. We call this method `s-kM`.

### 6.3.3 Re-embedding Copies

When using the clustering or community detection algorithms, only a partition of neighbors is computed, thus, we must first compute a set of faces in which to embed the copies. Then, we can route the paths between the copies and their neighbors.

To embed the copies we propose two alternatives, one algorithm based on minimizing Euclidean distances between copies and their neighbors, and one relying on minimizing crossings using paths in the dual.

For both methods, as described for ESN, we compute for each copy, the coordinate of a point inside the face it must be embedded in, and introduce a small wiggle to shift it away from that point while remaining inside the face.

#### Barycenter Positioning

To avoid long complex edge spanning the drawing, we investigate a straightfoward embedding method, by computing the barycenter position for each neighborhood partition. With this method we must be mindful of degenerate cases. Namely, for a partition with a single vertex, we do not place the copy on top of its neighbor but rather use the wiggle to shift it slightly away from that position. Similarly, for a partition with two neighbors, if the two neighbors are connected, the copy is not placed halfway between the neighbors but rather slightly shifted perpendicularly away from the halfway point. This ensures that the copy does not lie on the edge connecting its neighbors to one another. We then connect the copies to their neighbors using straight lines. This variant will be referred to as `e-B`.

#### Shortest Dual Face-path Method

Given a set of neighborhoods, the task is to compute a set of faces to embed the copies in, such that a copy's connection to its neighbors induces a minimum number of crossings. To do so, for the neighborhood $N(v)$ of a copy $v$, we loop over every face $f$ in the planarization $\Gamma_p$ of the input graph $\Gamma$, and compute the shortest path in the dual between $f$ and the faces incident to the vertices in $N(v)$. The length of the path in the dual corresponds to the number of faces that were crossed to connect the copy to its neighbors, which then tell us the number of crossings induced by the connection. We repeat this process over all faces to find the best face to position the copy in, and repeat the process for each neighbor partition. This algorithm is called `e-D` and is closely related with

`s-D`. It is computationally more efficient though as we only loop over one face at a time instead of $k$ faces.

This algorithm results in a collection of paths through the dual graph, between faces in which copies are embedded, and faces that are adjacent to the copy's neighbors. To route these edges, we begin by placing control points in each face that appears in one of those paths, using the method presented in Section 6.3.3. We then connect the control points along the paths using the same algorithm as for ESN, but allowing for a crossing to be created when connecting control points in two different faces, as shown in Fig. 6.5(4).

## 6.4 Evaluation

In this section, we present a two-fold evaluation of our pipeline. We first present a quantitative evaluation of our algorithms' performance with regards to the metrics introduced in Section 6.1.2 and secondly, we present a case study of four different drawings obtained with our pipeline. We identify the challenges of the vertex splitting approach, and outline some aesthetic considerations that result in higher quality drawings when using vertex splitting.

### 6.4.1 Dataset

While vertex splitting has previously been applied mostly to biological networks [WNSV19, NDG+17, NOM+19], our approach is meant to generalize its application. Therefore we focus instead on a less studied use case for vertex splitting, social networks [HBF08].

To evaluate our system, we use the graphs of the dagstuhl network graphs[2]. As our stated interested is the social networks, we use the *people graphs*, in which each node represents a researcher that has attended an event at dagstuhl and two people are connected if they both attended the same event. The repository contains four such graphs, one containing all events from 1990 onward, and three that only contain events occurring after 2001, 2011 and 2019 respectively.

As this work is meant to be a preliminary proof-of-concept for vertex splitting, we focus on smaller graphs. Therefore, to create our test graphs, for each of the four social networks, we filter the 20, 30 or 40 highest degree node. As solving vertex splitting in a disconnected graph is equivalent to solving smaller instances independently, we will add more nodes if necessary to ensure that the input graph is connected. Once we obtain a connected graph, we will remove the smallest degree nodes that do not disconnect the graph to ensure the sizes of the graph remain comparable.

[2] `https://github.com/mwallinger-tu/dagstuhl-network`

### 6.4.2 Experimental Setup

We implemented our bundling algorithm in python 3.9, using the networkx[3] library, and ran it on a machine with Windows 11 operating system, with an 2.40 GHz Intel i5-9300H CPU and 8G of RAM. The louvain and kmeans splitting algorithms were both implement using a specialized python library[4]. The ILP formulations were solved using the Gurobi optimizer[5].

As our use case considers automatically generated layout of large graphs, for each of the 16 input graphs described in Section 6.4.1, we obtained an input drawing using the Fruchterman-Reingold spring layout algorithm. We then ran each instance through the pipeline, using different algorithm combinations.

For each of the 16 instance we used the following three different vertex selection methods.

- First, we evaluted the perfomance of our algorithm when a single vertex should be split. We used `v-M` method to select the vertex involved in the highest number of crossings.
- Then, to emulate a more natural use case in which the vertex selection is done by a human, we used `v-M` to compute the five vertices that induce the largest number of crossings. We then randomly selected one of these vertices, removed it from the graph, and repeated the operation until $\frac{n}{10} + 1$ vertices were selected. We believe users are less likely to choose two vertices whose respective edges intersect the other's a lot, instead of choosing vertices which are independently problematic. Additionally, we believe users would not choose to split a large number of vertices to be split, thus we attempted to replicate that behavior with this method.
- Lastly, to evaluate instances of ESN, we generated plane input drawings by selecting vertices to be split using `v-HS`.

We note here that we did not allow the set of selected vertices to disconnect the graph, as we believe that a disconnected graph is not a desirable output. If an optimal solution to `v-HS` resulted in a disconnected graph, we first generated different embeddings for the drawing to avoid the situation, or would select a sub-optimal solution to prevent this situation.

For each of these 48 instances, we used every splitting algorithms (`s-FC`, `s-D`, `s-L` and `s-kM`). For the community and cluster detection algorithms, we used the `e-D` and `e-B` re-embedding methods respectively. As the Euclidean based clustering ignores the structure of the graph, it is a natural candidate for the Euclidean distance based embedding algorithm, and similarly, the graph community detection is more suited to

[3] `https://networkx.org/`
[4] `https://github.com/taynaud/python-louvain,https://scikit-learn.org/`
[5] `https://www.gurobi.com/`

the dual based re-embedding method. If the instances had not timed out after 1 hour (only relevant for `s-D` on larger instances), we measured their performance according to each of the metrics we defined Section 6.1.2.

Lastly, to compare clustering and community detection directly, we generated a set of larger graphs (50 nodes), and used both methods in conjunction with `v-HS` and `e-B`. We first ran the instances with the `s-L` algorithm. Then we ran the same instances using `s-kM`, and set the number of clusters to be computed to the number of splits that had been executed by `s-L`.

### 6.4.3 Quantitative Evaluation

Our main focus with this pipeline is crossing minimization, and these results are summarized in Fig. 6.6. We first notice that our algorithms were successful in reducing the number of crossings in the input drawings. The methods that were the least efficient were, `s-D` and `s-kM`. This is to be expected as they were only allowed one split operation per selected vertex. We can still observe that `s-D` was very successful in more sparse settings, when more vertices has been selected to be split. We can also see that its performance falls off significantly for larger instances. As expected, `s-FC` outperforms every other method, even for instances where the input drawing is not planar. Most interestingly, the Louvain based algorithm achieved strong results when compared to the optimal solution computed with `s-FC`. This seems to be linked to the fact that this method induced a large number of splits. Its performance appears to be inconsistent; we can observe that in one instance, it even resulted in a higher number of crossings than was present in the input. This is likely due to edges between copies of the same vertex intersecting one another. Lastly, we note that splitting a tenth of the vertices in the input graph produces good results, as even on larger graphs it appears it is sufficient to resolve about a third of the crossings in the input drawing. In general this seems to indicate that splitting a small number of vertices a large number should be a consideration for such a system.

With regards to resolving instances where the input is a planar graph, we summarize in Fig. 6.7 the performance of the `v-HS` algorithm. We observe that on graphs with only 40 vertices, half or more vertices have to be removed. As our implementation relies on an ILP, any non exact method for this setting would require an even larger number of vertices to be split. As our motivation is interactive human-in-the-loop systems, this indicates that ESN should not be one of the main targets as it significantly disturbs the input drawing.

We observe during the evaluation that the density of the graph seemed to have a strong impact on the performance of our algorithm. This is summarized in Table 6.1. This is natural as a larger number of edges results in a large number of crossings to resolve, and makes crossing resolution also more difficult. This means that unlike aggregation methods that are suited for denser drawings, vertex splitting is a strong candidate for larger sparser drawings. While the quality of the output was lower in denser drawings, we can observe in Fig. 6.8 that most algorithms were still very fast, even when splitting

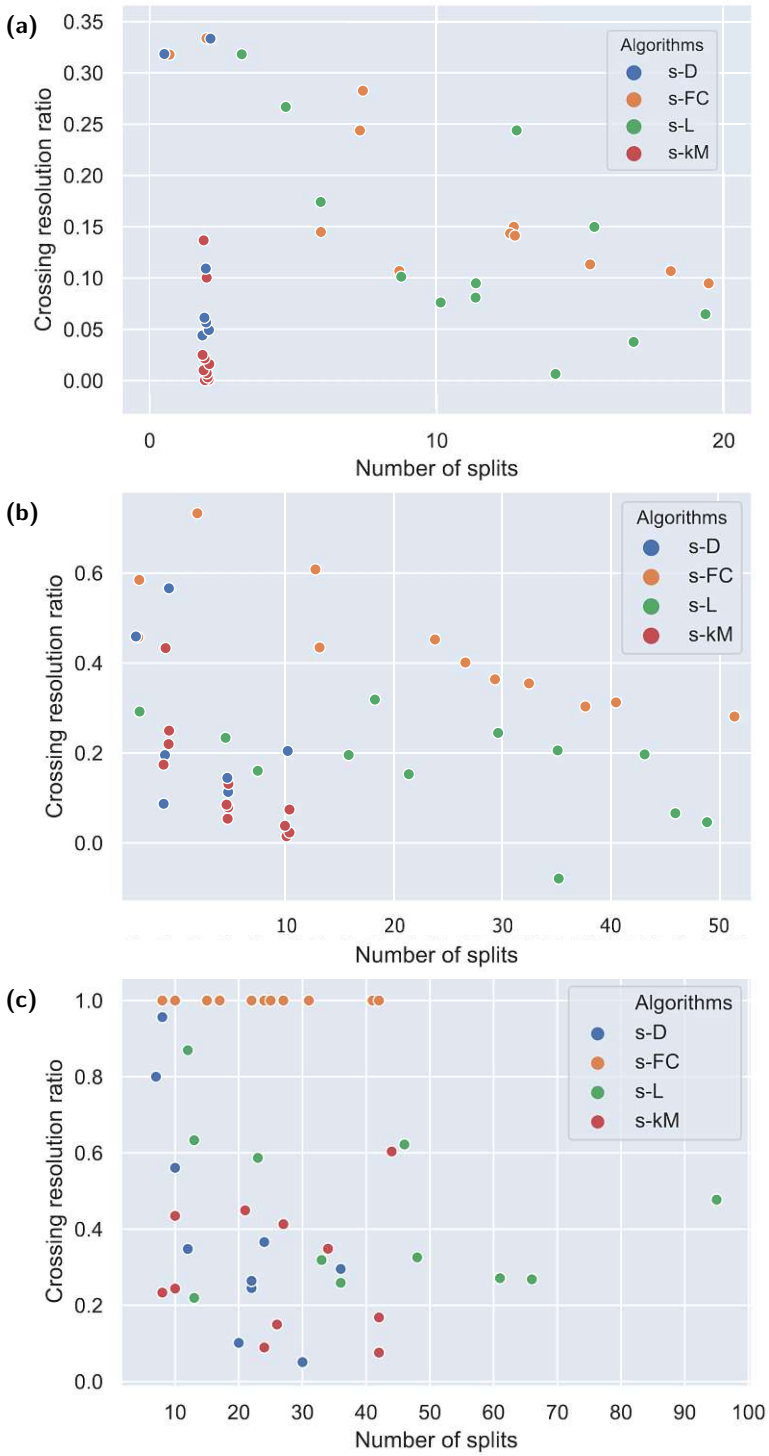



Figure 6.6: Crossing resolution ratio by the number of splits for each splitting algorithm, **(a)** vertices selected using `v-M` once, **(b)** using `v-U` **(c)** and using `v-HS`.

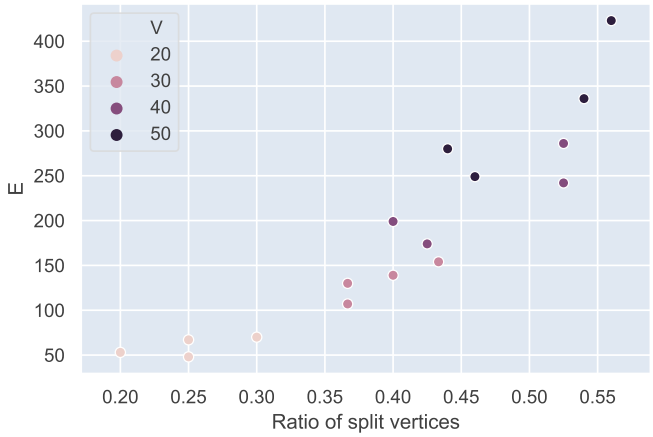



Figure 6.7: The ratio of vertices selected by `v-HS` by the total number of edges.

| | full | 2001 | 2011 | 2019 |
|---|---|---|---|---|
| density | 0.27 | 0.35 | 0.32 | 0.23 |
| cross resol ratio | 0.66 | 0.71 | 0.73 | 0.63 |
| number splits | 15.6 | 20.9 | 17.6 | 11.9 |
| runtime (sec) | 9.61 | 59.37 | 17.58 | 4.08 |

Table 6.1: Average performance for all algorithms over all metrics for each of the input graphs considered.

a lot of vertices, with the exception of `s-D`. This algorithm also timed out for all of the instances with 40 vertices and the two denser instances with 30 vertices (2001 and 2011). The high runtime that is sometimes incurred by `s-L` is linked to the edge routing computation, as Fig. 6.10 shows that it was significantly faster when paired with `e-D`.

We noted in Section 6.1.2 that complex curves were undesirable. Our algorithm's performance with regards to curve complexity is presented in Fig. 6.9. We see that when only a single vertex in the instance is split, long spanning edges are avoided by having copies with a small degree. Indeed, as only two copies are introduced by `s-D`, the copies have to cover a large subset of the neighborhood of the selected vertex, unlike the copies introduced by `s-FC`. Instead, as the neighbors of the copies created with `s-FC` are incident to the face in which the copy is embedded, we can often draw a straight-line edge to connect them. This reaffirms our initial observation, that a high number of splits of a small number of vertices is desirable.

Lastly, we consider the results of the comparison between `s-L` and `s-kM` in Fig. 6.10. As can be expected, considering the graph structure is more significant than considering the position of the neighbors of the split copies. Both algorithms are similarly fast, but for every input, `s-L` was able to resolve more crossings than `s-kM`.

In summary, we can see that splitting vertices is a successful technique to lower crossings

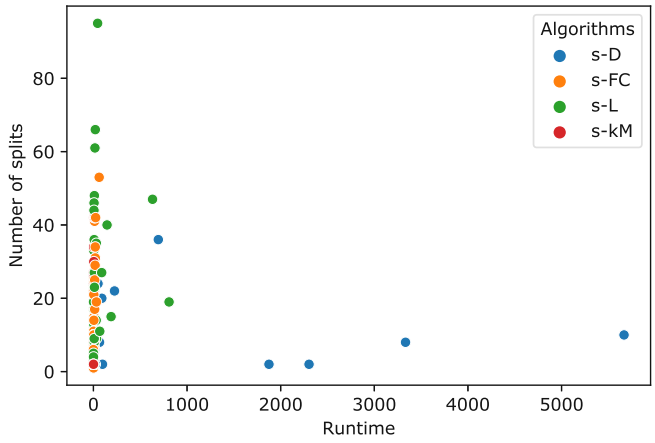



Figure 6.8: The runtime in seconds by the number of splits for each algorithm.

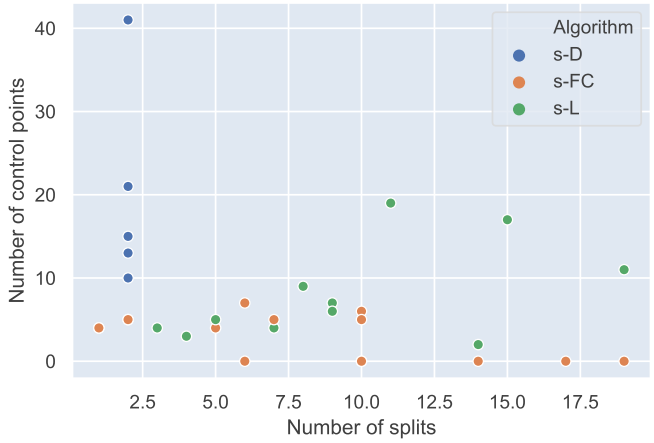



Figure 6.9: Number of control points generated by the number of splitting operations for each algorithm, when only a single vertex is selected to be split.

in graph drawings. We find that splitting a small number of vertices many times results in better performances as far as the quality metrics we have defined. We can also note that scalability is challenging. Denser graphs are not well suited to this approach, but sparse graphs lead to very positive results. Lastly, considering the underlying graph structure should be a focus when computing a neighborhood partition. Vertices in the same partition should be close with regards to graph distances to obtain good drawings.

### 6.4.4 Case Study

In this section we look closely at one of the instances from the previous section, shown in Fig. 6.11, to better understand the results from our algorithms. These drawings are created manually from the graph output by our pipeline using the same coordinates for

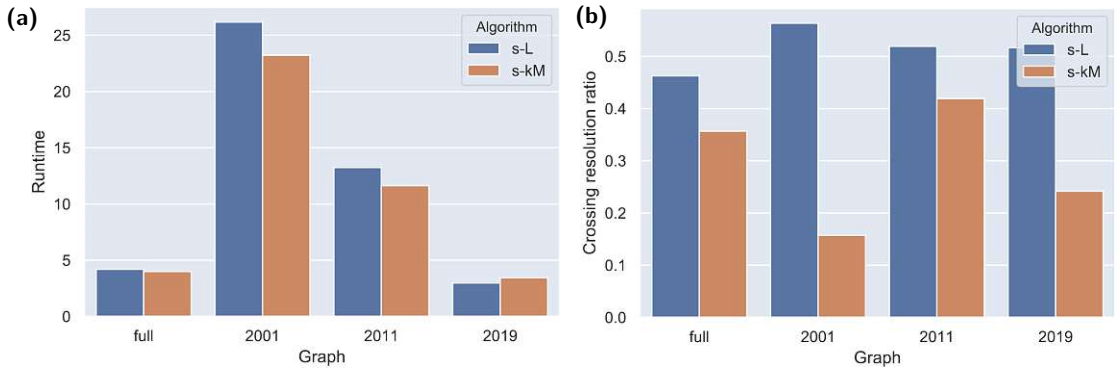



Figure 6.10: Performances of `s-L` and `s-kM` with regards to **(a)** runtime (in seconds) **(a)** and resolving crossings when vertices are selected with `v-HS` and embedding is computed with `e-B`.

vertices and control points. As we present a proof-of-concept in this chapter, creating renderings or post processing has not been implemented. Mainly, curves are currently polylines in the raw output.

Three vertices are selected by `v-HS` (Fig. 6.11(a)) and removed from the drawing. We can see in Fig. 6.11(b), that algorithm `s-D` performed well and only resulted in a single crossing remaining, even though it is limited to splitting a vertex only once. We also note that the orange vertex has not been split, since its complete neighborhood lies on the outer face, and the split operation was only used to change the embedding of the vertex rather than actually splitting it. Lastly, vertices embedded on the outer face tend to generate long curves. For example one of the blue edges is longer than the width of the drawing.

Similarly, `s-FC` (Fig. 6.11(c)) resulted in a large number of curves drawn in the outer face. While we consider an input drawing with straight line, we permit curves in the output. Curves have the benefit of highlighting which vertices are split, making them easier to find and potentially helping a user find paths through splits, but they add a lot of visual noise. Additionally, due to the small number of control points, they sometimes have sharp turn, as in Fig. 6.11(d).

While we noted that `s-L` (Fig. 6.11(d)) performed well in the previous section, we can notice here that it does not consider the input drawing or the coordinates of the vertices. The neighborhood assignment on the red vertices induces a crossing which could be avoided at no extra cost. Overall, it creates two more copies than `s-D` (Fig. 6.11(b)), and induces two more crossings, which makes `s-D` much more interesting for smaller instances as we had highlighted previously.

The drawing obtained using `s-kM` looks simpler due to the lack of curves, and the area it occupies has only marginally increased compared to the other drawings. We notice though, that because the lengths of the edges are minimized, the edges can become hard to read, and each embedding results in smaller faces in the drawing. Thus, when more

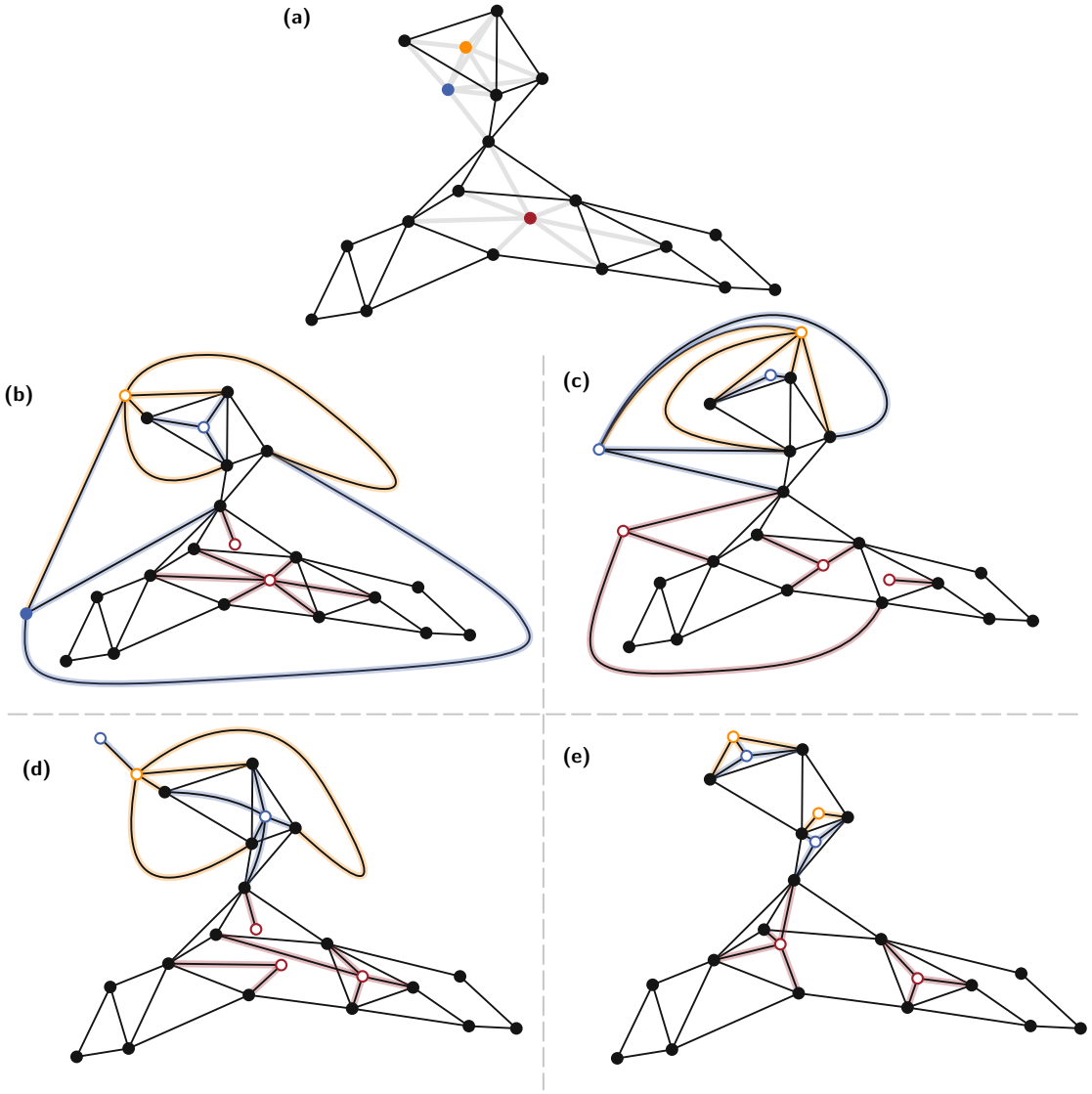



Figure 6.11: Graphs created using the output for the 2019 graph on 20 nodes, with the `v-HS` vertex selection algorithm, **(a)** shows the input drawing, with the colored vertices showing the vertices selected, **(b)** is obtained with `s-D`, **(c)** with `s-FC`, **(d)** with `s-L` and `e-D` and **(e)** with `s-kM` and `e-B`, and the copies are drawn with circles.

split occur we obtain very short edges and vertices being drawn too close to one another, which is undesirable as splitting is meant to help readability.

Overall, we can see that splitting is very effective on smaller graphs, but computing an effective embedding to reflect the splits is a delicate task, that is also dependent on the use case for the resulting drawing.

## 6.5 Limitation

Vertex splitting is a very complex problem that involves a large number of considerations. While we believe that we have identified a strong general guideline for splitting vertices for crossing minimization, the re-embedding sub-task has not been satisfyingly resolved. Different methods should be developed and implemented that could potentially mimic the drawing style of the input graph.

Similarly, our evaluation was limited to drawings obtained using a spring embedder, which does not accurately reflect the variety of existing drawing techniques. One should consider other inputs, including manually laid out graphs [NOM+19], or orthogonal graphs [WNSV19] which have been used in conjunction with splitting in the past.

While we only considered instances of a small size, it was sufficient to observe the challenges when it comes to scaling this approach to large graphs. Unlike graph simplification methods which reduce the amount of object being displayed, vertex splitting increases it. The use of vertex splitting for large graph is likely to be limited, though very specialized techniques for restricted settings could be considered, or different approaches that relied on the operation.

On the choice of vertices to be split, it is possible for a set of vertices to be selected that disconnect the graph. This is probably undesirable, especially since there is no guarantee that the output will be connected. While we did not allow for this situation to occur, having a set of vertices which cannot be split seems like a natural constraint for real world application.

Lastly, as our main interest is for vertex splitting to be extended to a human-in-the-loop system, an interactive tool should be designed, to better evaluate the impact of vertex splitting in graph drawing through a user study.

## 6.6 Chapter Conclusion

In this chapter we presented a preliminary proof of concept for vertex splitting for crossing minimization in drawings and found that it could efficiently reduce the number of crossings in a drawing. We introduced a flexible pipeline that can easily be adapted to other methods, or to user interactions as we decomposed the problems into simple and visual subproblems.

While we found that vertex splitting poses many challenges, we were able to identify general design guidelines for the problem. Namely, targeting planarity was not desirable as it required splitting a very large number of vertices. Instead, we found that focusing on a small number of vertices and splitting them many times was the most promising method. While we relied on an ILP to optimize this assignment, a greedy method would only result in smaller degree vertices which do not induce a lot of clutter. For this reason as well, we believe that vertex explosions should be considered for this problem.

One of the largest remaining question to understand is to the task of embedding copies back into the graph. In general when attempting to reduce crossings, straight-line edges are not desirable. Since a large number of layout algorithms produce straight-line drawings, this creates a conflict, as the input's drawing conventions do not match the ones of the re-embedding system. A simple solution to this is to consider very low degree vertices, as the task is almost trivial for them.

# Part II

# Word Clouds

CHAPTER 7

# Layered Semantic Word Clouds

From a theoretical point of view, semantic word clouds are an extension to the contact graph problem. Contact representations of graphs are a well-studied topic in graph theory, graph drawing, and computational geometry [Fel13, Koe36, dFdMR94]. Vertices are represented by geometric objects, e.g., disks or polygons, and two objects touch if and only if they are connected by an edge. They find many applications, for instance in VLSI design [YS93], cartograms [NK16]. In this chapter, we will focus on the model described for semantic word clouds [WPW+11, BFK+14]. This chapter is based on joint work with Martin Nöllenburg and Jules Wulms, and was presented at GD 2021 [NVW21]. An extended abstract of this chapter was also presented at EuroCG

Word clouds, in which words have different font size and are packed without semantic context such as the one shown in Fig. 7.1, have received criticism on their readability. The audience sometimes fails understand the sometimes fails at understanding the underlying


Figure 7.1: Word cloud generated from the apnews.com frontpage by worditout.com on the day of the certification of Joe Biden's victory in the 2020 US elections.

data (while enjoying their playful nature) [HPP+20]. For example, neighboring words that are not semantically related can be misleading (see marked words in Fig. 7.1). As a way to improve readability, *semantic* word clouds have been introduced [CWL+10, WPW+11, BFK+14]. In semantic word clouds, an underlying edge-weighted graph indicates the semantic relatedness of two words, whose positions are chosen such that semantically related words are next to each other while unrelated words are kept far apart.

Classic word clouds are often generated using forced-based approaches, alongside with a spiral placement heuristic [VWF09, WCB+18, WCZ+20] that allows for a very compact final layout. This method is powerful even when the rough position of a word is dictated by an underlying map [BCL+16, LDY18]. Semantic word clouds on the other hand have been approached with many different techniques, e.g., force directed [CWL+10], seam-carving [WPW+11], and multidimensional scaling [BKP14]. The problem has also been studied from a theoretical point of view, where an edge of the semantic word graph is realized if the bounding boxes of two related words properly touch; the realized edge weight is gained as profit. Then the semantic word cloud problem can be phrased as the optimization problem to maximize the total profit. Barth et al. [BFK+14] and later Bekos et al. [BvDF+17] gave several hardness and approximation results for this problem (and some variations) on certain graph classes. The underlying geometric problem also has links to more general contact graph representation problems, like rectangular layouts [BGPV08] or cartograms [vKS07].

In most of the literature about layered graphs, vertices are assigned to rows without a predefined left-to-right order, yet this has interesting properties in the context of word clouds. For instance, layered rectangle contact representations are compact, assuming a good assignment they have an even distribution of words and our eye naturally understands words grouped in rows or tables. In this chapter we therefore study row-based contact graphs of unit-height but arbitrary-width rectangles, which may represent the bounding boxes of words with fixed font size.

**Structure of the chapter:** We start by studying the classic word cloud problem of minimizing the drawing area, we introduce in Section 7.2 a flow network based area minimization algorithm for the layered word cloud problem. We then focus on the semantic quality of the layout, and study the contact maximization optimization version of our problem. We show in Section 7.3 that we can find a layout of the rectangles for two-layer drawings that maximizes contacts, and present an integer linear programming model for the $n$ layer setting.

## 7.1 Preliminaries

As input we take a layered graph $G = (V, E)$ on $L$ layers, with an arbitrary number of vertices per layer. Each vertex $v_{i,j} \in V$ is indexed by its layer $i \in [0, L-1]$ and its position $j$ within the layer: $v_{i,j}$ is the $j^{th}$ vertex on the $i^{th}$ layer. The edge set $E$ consists of edges connecting each vertex $v_{i,j}$ to its neighbors $v_{i,j-1}$ and $v_{i,j+1}$ on the same row

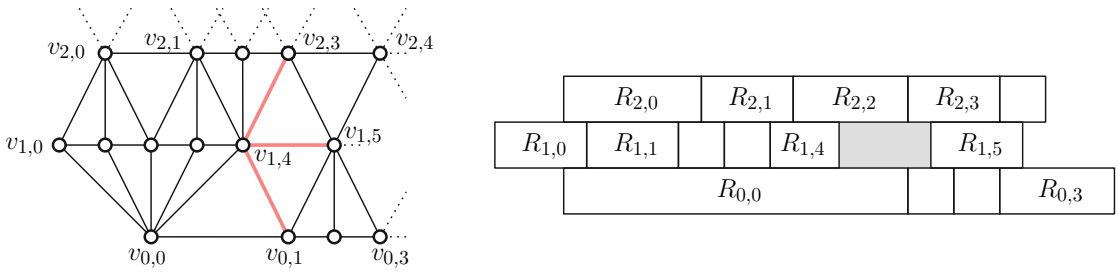



Figure 7.2: Partial drawing of a graph $G$, along with a representation $\mathcal{R}$ of the visible vertices of $G$. Red edges are not realized, due to the gray gap in $\mathcal{R}$.

Figure 7.3: Allowing false adjacencies (red) could reduce lost adjacencies (blue).

(if they exist), and connections between adjacent rows form an internally triangulated graph. We associate each vertex with an axis-aligned unit-height rectangle $R_{i,j}$ with width $w_{i,j}$, and $y$-coordinate $i$. We want to compute its $x$-position $x_{i,j}$ given by the $x$-coordinate of its bottom left corner such that the rectangles do not overlap except on their boundaries (see Fig. 7.2). Leaving whitespace between two rectangles on the same layer is allowed and forms a *gap*. Such a layout $\mathcal{R}$ is called a *representation* of $G$. An edge $(u, v) \in E$ is *realized* in a representation $\mathcal{R}$ if rectangles $R_u$ and $R_v$, representing vertices $u$ and $v$, intersect along their boundaries for a positive length $\varepsilon > 0$, which we denote by $(R_u, R_v) \in \mathcal{R}$. If $R_u$ and $R_v$ are horizontally adjacent we call the contact a *horizontal contact*.

Otherwise, the intersection is located along a horizontal boundary if $R_u$ and $R_v$ are on adjacent layers; these are called *vertical contacts*. Contacts between rectangles whose vertices are not adjacent in $G$ are *false adjacencies*. Such adjacencies can mislead a user to infer a link between unrelated words, invalidating the representation. Within this model we study two problem variations, *area minimization* and *contact maximization*.

For the area minimization problem the goal is to produce a representation $\mathcal{R}$ that minimizes the total width of the gaps in $\mathcal{R}$. The contact maximization problem asks to maximize the number of adjacencies realized in $\mathcal{R}$, as specified by edge set $E$. For both optimization criteria, false adjacencies are forbidden: otherwise a trivial gap-less representation would always be a solution to the area minimization problem and in the case of contact maximization, false adjacencies may reduce the number of lost contacts with respect to a valid optimal solution as Fig. 7.3 shows. We say a representation is *valid* if it has no false adjacencies.

## 7.2 Area Minimization

To solve the area minimization problem, we construct a flow network $N = (G' = (V', E'); l; c; b; cost)$ for a given vertex-weighted layered graph $G = (V, E)$, with edge capacity lower bound $l\colon E' \to \mathbb{R}_0^+$, edge capacity $c\colon E' \to \mathbb{R}_0^+$, vertex production/consumption $b\colon V' \to \mathbb{R}$ and cost function $cost\colon E' \to \mathbb{R}_0^+$. Each unit of cost will represent a unit length gap and each unit of flow on an edge will represent a unit length contact. To build the network we create two vertices $v^a$ and $v^b$ for each rectangle, that respectively receive the flow from the lower layer and output flow to the upper layer, and one for each potential gap, located between each sequential pair of rectangles in the same layer. Every edge $e$ that ends on a gap vertex has $cost(e) = 1$. We also add an edge $e$ between $v^a$ and $v^b$ for each $R_{i,j}$ with $l(e) = c(e) = w_{i,j}$ and no cost to ensure that rectangle nodes receive exactly as much flow as they are wide.

The intuition behind the network is that it represents a stack of layers consisting of rectangles and gaps, with a maximum width of $w_{\max} \cdot K$, $K$ being the maximum number of rectangles per layer, and $w_{\max}$ the width of the widest rectangle. To facilitate this flow on all layers, there are buffer vertices on both sides of each layer. Each rectangle is as wide as the amount of flow its vertices $v^a$ and $v^b$ receive, and has contacts with its upper and lower neighbors as wide as the flow on the edges representing these contacts. Every vertex has edges to the layer above as far as its rectangle is allowed to have contacts: a rectangle $R_{i,j}$ that has only one upper neighbor, will have an edge to that neighbor, and to the gaps on that neighbor's right and left side. Any further edge would be to another rectangle with which $R_{i,j}$ should not share a contact, and such edges would hence result in false adjacencies. We picture the stack bottom-up, meaning that the flow comes in at the bottom layer and exits from the top layer. A gap block $g_{i,j}$ will reach as far left and right as its left and right neighbors in the same row: if rectangle $R_{i,j}$ lies directly left of $g_{i,j}$, then the furthest left upward neighbor of $R_{i,j}$ is the furthest left upward neighbor of $g_{i,j}$. If $g_{i,j}$ could reach even further, then it would essentially push $R_{i,j}$ into a false adjacency. The construction is sketched in Fig. 7.4.

**Flow Network Construction:** For each $v_{i,j} \in V$ we create two copies $v^a_{i,j}$ and $v^b_{i,j}$ in $V'$; for each pair $v_{i,j}, v_{i,j+1}$ we introduce a gap vertex $g_{i,j}$, for each layer a left and right buffer vertex $l_i$ and $r_i$ and a global source and sink, $s$ and $t$. All the vertices $v$ have $b(v) = 0$ except $b(s) = w_{max} \cdot K$ and $b(t) = -w_{max} \cdot K$ with $w_{max}$ the width of the widest rectangle and $K$ the maximum number of rectangles per layer.

Unless stated otherwise, each edge $e$ has $c(e) = \infty$, $l(e) = 0$ and $cost(e) = 0$. The rectangle-rectangle edges for two vertically adjacent vertices $u$ and $v$ are from the $u^b$ vertex on layer $i$ to the $v^a$ vertex on layer $i + 1$. For each $v^a_{i,j}, v^b_{i,j}$ pair in $G'$ we add an edge $e_{i,j}$ from $v^a_{i,j}$ to $v^b_{i,j}$ with $c(e_{i,j}) = w_{i,j}$ and $l(e_{i,j}) = w_{i,j}$. This ensures that a vertex must receive exactly as much flow as the width of the rectangle it represents. We also add edges going in and out of the buffer vertices: to minimize the gaps it is preferable for the outer rectangles to get missing flow from the buffers on the outside or to route

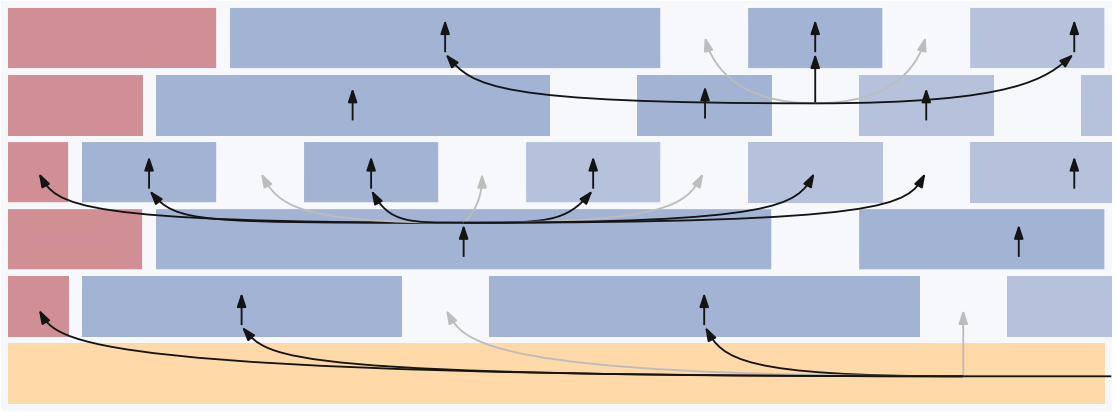


Figure 7.4: Parts of a flow network: outgoing edges from the source (orange), a rectangle (blue) and a gap; red buffer rectangles; gray edges have cost 1.

excess flow to these buffers.

Edges from rectangle vertices to gap vertices are defined as follows. For a vertex $v^b_{i,j}$, we add an edge to all vertices $v^a$ representing its neighbors in $G$ on layer $i+1$, and we add edges to the gaps left and right of those neighbors. All edges to gaps have $cost = 1$, to penalize the creation of gaps. Lastly we add edges from gaps to other vertices. For a gap vertex $g_{i,j}$ we look at its left and right neighbors $v^b_{i,j}$ and $v^b_{i,j+1}$ respectively. We add edges from $g_{i,j}$ to all rectangles and gaps that have an incoming edge from $v^b_{i,j}$ and $v^b_{i,j+1}$, and assign $cost = 0$ for edges into rectangles, and $cost = 1$ for edges into gaps. We do not penalize the outgoing flow of a gap, since we already count the incoming flow.

From this network we can easily deduce that the minimum cost flow corresponds to the solution that minimizes total gap length: only gap vertices have a cost, hence the flow avoids these vertices whenever possible. We can construct the network and then solve the minimum flow problem in polynomial time. In the obtained solution, the flow values found on each edge should represent the length of the overlap between these elements, which allow us to construct the corresponding representation. However, in practice the relationship between units of flow and contact length is not always direct. Indeed, consider a 4 vertex configuration where $a$ lies bottom left, $b$ bottom right, $c$ top left and $d$ top right, and both $a$ and $b$ have an edge to both $c$ and $d$. Note that in such a case $a$ and $d$, or $c$ and $b$, would be gaps. It is possible that $a$ favors sending its flow to $d$ and $b$ to $c$, which is impossible to represent as contacts in a configuration of rectangles.

Once the flow has been computed, we can locally swap the required flow between the crossing edges to resolve those *crossing patterns*.

**Theorem 15.** *Given a graph $G = (V, E)$, the cost of a minimum-cost flow $f$ in $N$ equals the minimum total gap length of any valid representation of $G$. An area-minimal representation of $G$ is constructed from $f$ in polynomial time.*

*Proof.* Given any graph $G = (V, E)$, the associated network $N$ has production equal to its consumption $\sum_{v \in V'} b(v) = b(s) + b(t) = 0$. The source produces $b(s) = w_{max} \cdot K$ flow, which is available to every vertex on layer 1 (except $v^b$ vertices whose incoming neighbor is $v^a$ on the same layer). Any vertex $v^a_{1,j}$ must receive $w_{1,j}$ units of flow, as its only edge towards $v^b_{1,j}$ has capacity constraint $c = l = w_{1,j}$. Because $w_{1,j} \leq w_{max}$ and since there are at most $K$ vertices $v^a_{1,j}$ on layer 1, the capacities can be satisfied. Any edge in $E'$ goes from layer $i$ to $i + 1$, except for $(v^a_{i,j}, v^b_{i,j})$, but the exact amount of flow that comes from layer $i - 1$ into $v^a_{i,j}$ will go through $v^b_{i,j}$ to layer $i + 1$. Hence for the same reason as in layer 1, there is enough flow to satisfy the edge capacity constraints, while excess flow is routed through (0-cost) buffers or (1-cost) gaps.

Since in this network only flow that goes into a gap vertex has a (non-zero) cost, flow into gap vertices, and therefore also the total gap width, is minimized. The minimum cost procedure finds this optimal flow $f$.

We construct a minimum-area representation $\mathcal{R}$ by placing rectangles row by row: we leave buffers and gaps equal to the flow routed through the corresponding vertices, and align all rows on the left. The total width of each row, including buffers and gaps, is $w_{max} \cdot K$. Since the flow through each rectangle is constant, the area of the buffers is maximized, to minimize the area occupied by gaps. □

While this method minimizes the area occupied by the drawing it will not always lead to the representation with the minimum bounding box. To minimize the size of the bounding box, we propose to limit the amount of flow outgoing from the source node and incoming into the sink node. If the chosen bound is too small then the flow network will not be realizable. We can thus perform a binary search between the width of the longest layer $W_{\max}$ as a lower bound and $w_{\max} \cdot K$ as an upper bound. Assuming input widths have integer values, this method would add a $\mathcal{O}(\log(w_{\max} \cdot K - W_{\max}))$ factor to the flow runtime.

## 7.3 Maximization of Realized Contacts

In this section we propose algorithms that maximize the number of realized contacts. We start with a linear-time algorithm for $L = 2$, followed by an integer linear programming model for $L > 2$. The complexity for $L > 2$ remains open.

### 7.3.1 Linear-time Algorithm for $L = 2$

In this section we describe an algorithm $\mathcal{A}$ for the case where the input has 2 layers. On a 2-layer graph, a vertex either has one neighbor in the adjacent layer, or more than one. If a vertex has one neighbor we call it a *T-vertex* and if there are more neighbors, it is called a *fan*. A *block* is a maximal sequence of consecutive rectangles in a layer $i$, for which each horizontal contact is realized. A block from the $j$th until the $l$th vertex of row $i$ is the sequence $(R_{i,j}, \ldots, R_{i,l})$, where for each $k \in [j, l-1]$ holds that $(R_{i,k}, R_{i,k+1}) \in \mathcal{R}$.

In a given 2-layer graph, there will always be a layer that starts on a fan, while the opposite layer starts on a T-vertex. Assume without loss of generality that $R_{0,0}$ is a fan, otherwise swap the two rows for the duration of the algorithm. Algorithm $\mathcal{A}$ starts by placing $R_{0,0}$, followed by all its neighbors on the adjacent layer, from left to right, ending with $R_{1,j}$. Rectangle $R_{1,j}$ is again a fan, and the process of placing all opposite-row neighbors, left to right, is repeated for $R_{1,j}$ and every consecutive fan, as they are encountered. We call this placement ordering $\prec$.

When we add a rectangle $R_i$ (fan or T-vertex), we always first attempt to add it next to its horizontal predecessor, if possible (no false adjacency). Though, if the horizontal predecessor is too far left, we place $R_i$ in the leftmost allowed position. Let $R_0$ be the first rectangle in $\prec$, which is placed on position $x_0$ by $\mathcal{A}$. Algorithm $\mathcal{A}$ then proceeds by adding $R_1$, representing a T-vertex in the opposite row. Rectangle $R_1$, with width $w_1$, is placed leftmost, on coordinate $x_0 + \varepsilon - w_1$. We then proceed to add all rectangles corresponding to other T-vertices of $R_0$ one by one, such that all horizontal contacts are realized. Once a T-vertex $R_i$ cannot reach $R_0$, we store the number of contacts currently realized by $R_0$ as well as its position $x_0$, and slide $R_0$ rightward, to the leftmost position $x_0'$ that allows a contact of $\varepsilon$ with $R_i$. Note that, since we placed $R_1$ in the leftmost position that allowed a contact of width $\varepsilon$ with $R_0$, we lose at least one contact by moving $R_0$ rightward. If placing $R_0$ at $x_0'$ ties the number of contacts of $x_0$, then we set $x_0 \coloneqq x_0'$. If $x_0'$ is strictly worse, then the representation is reset to having $R_0$ at $x_0$. From that point on, every time we add a new rectangle, we attempt this shift of the fan and update the position when we find a tie or when we realize more contacts. We repeat this operation for each rectangle, following the order $\prec$, always shifting the last encountered fan.

However, once we consider a fan $R_f$ that is not $R_0$, any sliding operation will be attempted on the block containing $R_f$, rather than just $R_f$. As before, we always shift the block to the leftmost position that realizes the contact between $R_f$ and the newly placed rectangle. We remember position $x_f$ that realizes most contacts, and favor the newest position on a tie. In case moving the block containing $R_f$ leads to strictly less contacts, we also try to move only $R_f$ instead. This starts a new block containing just $R_f$.

**Theorem 16.** *Algorithm $\mathcal{A}$ computes a contact maximal valid representation with contacts of length at least $\varepsilon$ for a given 2-layer graph $G$ in linear time.*

*Proof.* We show that during the placement of rectangles by algorithm $\mathcal{A}$, the invariant holds that a representation computed for the first $n$ rectangles in $\prec$ achieves a maximum number of contacts. First observe that, we start by placing $R_0$ and $R_1$, such that their only contact is realized. The invariant therefore holds at the start.

Now assume that the invariant holds after $\mathcal{A}$ placed $n-1$ rectangles, such that the current representation $\mathcal{R}^*$ is a contact maximal representation of the first $n-1$ rectangles in $\prec$, and realizes $k$ contacts. Algorithm $\mathcal{A}$ now adds the next rectangle $R_n$. We show that the new representation is contact maximal.

The maximum number of contacts that the new representation can realize is $k+2$, because rectangle $R_n$ can achieve at most one vertical and one horizontal contact. If there was a representation of the $n$ rectangles that realized $k' > k+2$ contacts, then removing $R_n$ would leave $k'-2 > k$ contacts, which contradicts our assumption that $\mathcal{R}^*$ is optimal. If $R_n$ naturally realizes its vertical and horizontal contacts, the representation realizes $k+2$ contacts. There is no configuration where we cannot realize any contact when adding rectangle $R_n$ to $\mathcal{R}^*$, since one of the rows extends further, and hence $R_n$ either can achieve its horizontal contact, or the opposing row extends at least $\varepsilon$ and the vertical contact can be realized. We therefore consider only the cases where adding $R_n$ results in a representation with $k+1$ contacts. Throughout the rest of the proof, we refer to the vertical fan neighbor of $R_n$, that has already been placed, as $R_f$. $R_f$ necessarily lies in the opposite layer. The only rectangles that will be moved when placing $R_n$ are the rectangles in the block containing $R_f$. Any other rectangles and adjacencies are untouched. So it is sufficient to count the adjcencies gained and lost by the block containing $R_f$ and $R_n$ to confirm that the representation of $n$ rectangles is optimal.

- If $R_n$ realizes only its vertical contact with $R_f$, then rectangle $R_l$, neighboring $R_n$ on the left, is necessarily a fan (see Fig. 7.5a)). Assume for contradiction that $R_l$ is a T-vertex. This implies that $R_f$ is $R_l$'s only allowed vertical contact. Since $R_n$ achieves contact with $R_f$ and $R_f$ is leftmost, $R_l$ must also be in contact with $R_f$. As a result the horizontal contact with $R_l$ always happens, contradicting our assumption that $R_n$ has only a vertical contact. As a consequence, $R_l$ is a fan, and the position for a fan is the position that maximizes contacts, favoring newer positions (or positions more to the right) for ties. An alternative to $\mathcal{R}^*$ with $R_l$ in contact with $R_f$ must therefore achieve less than $k$ contacts. Additionally, $R_n$ cannot be slid left of $R_f$ even though there is a gap towards $R_l$, as the rectangles left of $R_f$ are not part in the neighborhood of $R_n$. Thus, sliding $R_n$ further left would cause false adjacencies. There is no contact maximal representation that preserves the contact between $R_l$ and $R_n$, and hence $k+1$ contacts is maximal.

- If $R_n$ realizes only its horizontal contact, then it fails the contact with $R_f$. We distinguish between $R_n$ being a fan or a T-vertex.

  - If $R_n$ is a T-vertex, then $R_f$ is too far to the left, and the algorithm will try to create representations where $R_f$ is in the leftmost position that realizes a contact with $R_n$: either by moving the block containing $R_f$, or alternatively moving just $R_f$.

    * If $R_f$ remains in its original position, which does not achieve the vertical contact with $R_n$, it means that the leftmost position of $R_f$, that achieves the contact with $R_n$ realizes strictly less contacts than the original position of $R_f$ in $\mathcal{R}^*$. As a result, there is no contact maximal representation that preserves the contact between $R_f$ and $R_n$, and thus $k+1$ contacts is optimal (see Fig. 7.6a).

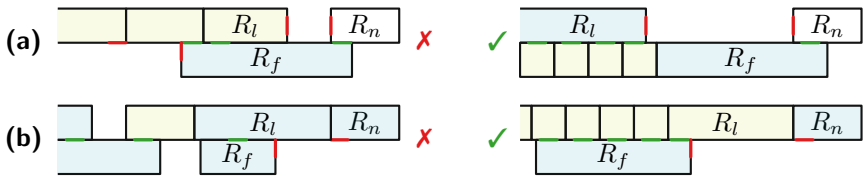



Figure 7.5: Two configurations where $R_l$ is necessarily a fan (blue) or a T-vertex (yellow). **(a)** If $R_n$ can only achieve a vertical contact, $R_f$ is a fan. **(b)** If $R_n$ can only achieve a horizontal contact and is a fan, $R_f$ is a T-vertex.

  - ∗ When the block containing $R_f$ is shifted to the right to create the alternative representation that ties the number of contacts of $\mathcal{R}^*$, at most one vertical contact must have been lost, to gain the vertical one. Otherwise, $R_f$ can give up its horizontal contact to achieve the same result (see Fig 7.6b). Thus, while $R_n$ realises 2 contacts, $R_f$ lost one and the representation realizes $k+1$ contact. This alternative representation is optimal and used by $\mathcal{A}$ instead of the representation where $R_f$ is not moved rightward.
  - ∗ If the block containing $R_f$ does not lose contacts when shifted, then the overall representation will gain the contact between $R_n$ and $R_f$ contact. This configuration is optimal because it achieves $k+2$ contacts (see Fig. 7.6c).
- If $R_n$ is a fan, then rectangle $R_l$, again neighboring $R_n$ on the left, must be a T-vertex, since $R_f$ cannot be an empty fan (see Fig. 7.5b). Assume for contradiction that $R_l$ is a fan, then $\mathcal{A}$ would try to realize the vertical contacts between $R_f$ and the fans $R_l$ and $R_n$ on the other row. In this case $R_f$ would necessarily be an empty fan, and hence $\mathcal{A}$ will try to sacrifice the contact with the left neighbor of $R_f$, to gain the vertical contact between $R_f$ and $R_n$. As there are no T-vertices for $R_f$ to lose contacts with, $\mathcal{A}$ will always find at least a tie between the configuration that has the vertical contact between $R_f$ and $R_n$, and $\mathcal{R}^*$, and hence prefer the new configuration. This contradicts our assumption that $R_n$ realizes only its horizontal contact, and hence $R_l$ must be a T-vertex. When $R_n$ is added, an alternative representation is created with $R_f$ shifted to the leftmost position that realizes the contact with $R_n$. Since this configuration is not chosen by $\mathcal{A}$, the configuration realizes less than $k+1$ contacts, and therefore $k+1$ contacts is optimal.

Thus, after adding $R_n$ to $\mathcal{R}^*$, algorithm $\mathcal{A}$ produces a representation that realizes $k+2$ contacts, if it is possible, and $k+1$ otherwise, which is optimal. The invariant will therefore still be true after $\mathcal{A}$ added all rectangles, producing a contact maximal representation of our input graph.

The algorithm considers each rectangle either only once, or as many times as its degree when an alternative shifted representation is created. As the input graph is planar, the

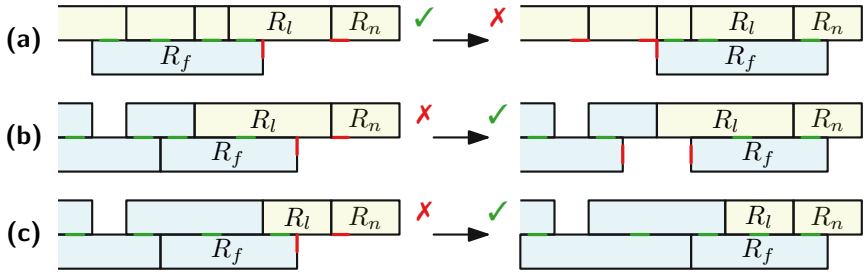



Figure 7.6: Three configurations where T-vertex $R_n$ does not realize vertical contacts with $R_f$ initially. We move $R_f$ and either **(a)** reset if the number of contacts is strictly worse, or save when we find **(b)** a tie, or **(c)** an increase in contacts.

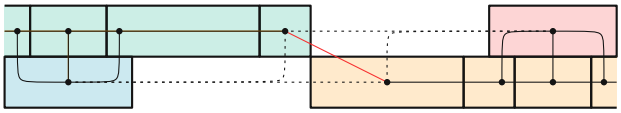


Figure 7.7: Edge case where the prefered position of the fan vertices (red and blue) causes a point contact on fan vertices (orange and green)

algorithm runs in linear time. □

As one may already have realized, in some cases, when looking at two sequential fans that are not in contact, two T-vertices will be in a point contact. This happens when the T-vertices cannot overlap without creating a false adjacency, as we show in see Fig 7.7.

### 7.3.2 ILP

To solve the contact maximization problem on $L > 2$ layers we propose an ILP formulation, which intuitively works as follows. We create a binary contact variable $c(e)$ for each edge $e$ in the input graph. If a contact is not realized, we set $c(e) = 1$ to satisfy the position constraints, otherwise we can set $c(e) = 0$. To handle false adjacencies we add for each rectangle a constraint on the first false contact that happens from the right and left on the row above, if they exist. We use hard constraints on the rectangle coordinates to prevent the false adjacencies. The objective is to minimize the sum over all contact variables, under all these constraints, to maximize the number of realized contacts in a solution.

Our integer linear programming formulation $M$, which can be used to find optimal solutions to the contact maximization problem is as follows. In this formulation, the constants are the width $w_{i,j}$ of each rectangle, the list $E$ of contacts, the minimal length $\varepsilon$ of a contact and a large integer $M$ for indicator type constraints. Additionally, we have the lists $F_L$ and $F_R$ of false adjacencies, defined as follows. Consider the rectangle representing vertex $v_{i,j}$ and its neighbors $N = \{v_{i+1,k}, \ldots, v_{i+1,l}\}$ on the row above. For each such vertex $v_{i,j}$, we store the pairs $(v_{i,j}, v_{i+1,k-1}) \in F_L$ and $(v_{i,j}, v_{i+1,l+1}) \in F_R$, if $v_{i+1,k-1}$ and $v_{i+1,l+1}$ exist. These pairs represent the false adjacencies of $R_{i,j}$ with the

rightmost rectangle left of $N$ and the leftmost rectangle right of $N$, respectively. Note that preventing these two false adjacencies will prevent all false adjacencies of $R_{i,j}$ with rectangles in row $i+1$, as long as the order of the rectangles in both rows is correctly maintained.

The variables of the ILP are the following. We use $x_{i,j}$ as variables for the position of each rectangle, representing the bottom left corner of the rectangle, and Boolean variables $c(v, v')$ which indicate that a contact is not realized between $v$ and $v'$. We start by stating the optimization function, which minimizes the sum of all $c$ variables. In practice, this means we maximize realized adjacencies.

$$\text{minimize} \sum_{(v,v')\in E} c(v, v') \tag{7.1}$$

The following inequalities ensure firstly, that there is no overlap between rectangles on the same layer (2), and secondly, check whether the horizontal contact is realized (3): if $x_{i,j}$ is too small, then $c$ must be set to 1, and hence the represented rectangle is too far left to have the contact naturally.

$$x_{i,j} + w_{i,j} \leq x_{i,j+1} \qquad \forall (v_{i,j}, v_{i,j+1}) \in E \tag{7.2}$$
$$x_{i,j+1} \leq x_{i,j} + w_{i,j} + c(v_{i,j}, v_{i,j+1})M \qquad \forall (v_{i,j}, v_{i,j+1}) \in E \tag{7.3}$$

The next inequalities ensure that the contacts between rectangle $R_{i,j}$ and all of its neighbors on layer $i+1$ are realized, such that the left side of a neighbor is left of the right side of $R_{i,j}$, or symmetrically, the right side of a neighbor is right of the left side of $R_{i,j}$. Again, if we cannot satisfy the inequality with $c = 0$, then it is set to 1, and since $M$ is a large value, the inequalities are trivially satisfied.

$$x_{i+1,j'} \leq x_{i,j} + w_{i,j} - \varepsilon + c(v_{i,j}, v_{i+1,j'})M \qquad \forall e(v_{i,j}, v_{i+1,j'}) \in E \tag{7.4}$$
$$x_{i,j} \leq x_{i+1,j'} + w_{i+1,j'} - \varepsilon + c(v_{i,j}, v_{i+1,j'})M \qquad \forall e(v_{i,j}, v_{i+1,j'}) \in E \tag{7.5}$$

Finally, we model false adjacencies using the pairs stored in $F_L$ and $F_R$. For a pair in $F_L$, the rectangle $R_{i+1,j'}$, with which $R_{i,j}$ has a false adjacency, should stay left of $R_{i,j}$. This forces the left side of $R_{i,j}$ to be right of the right side of $R_{i+1,j'}$. Symmetrically, pairs in $F_R$ prevent false adjacencies when $R_{i+1,j'}$ is to the right of $R_{i,j}$.

$$x_{i+1,j'} + w_{i+1,j'} \leq x_{i,j} \qquad \forall (v_{i,j}, v_{i+1,j'}) \in F_L \tag{7.6}$$
$$x_{i,j} + w_{i,j} \leq x_{i+1,j'} \qquad \forall (v_{i,j}, v_{i+1,j'}) \in F_R \tag{7.7}$$

**Theorem 17.** *Solving ILP model $M$ optimally, results in an optimal solution for the contact maximization problem.*

*Proof.* First, we show that whenever a variable $c$ is set to zero, then a contact is realized. Let us assume that $c = 0$, then one of the following two cases applies.

- If $c(v_{i,j}, v_{i,j+1}) = 0$, then the horizontal contact between $R_{i,j}$ and $R_{i,j+1}$ is realized. Since $x_{i+1,j} \leq x_{i,j} + w_{i,j}$ and $x_{i,j} + w_{i,j} \leq x_{i,j+1}$, then $x_{i,j} + w_{i,j} = x_{i,j+1}$. Thus the coordinate of the right side of $R_{i,j}$ lies on the left side of $R_{i,j+1}$, and there is a horizontal contact.
- If $c(v_{i,j}, v_{i+1,j'}) = 0$, then the vertical contact between $R_{i,j}$ and $R_{i+1,j'}$ is realized. We have both $x_{i+1,j'} \leq x_{i,j} + w_{i,j} - \varepsilon$ which means that the left side of $R_{i,j}$ lies left of the right side of $R_{i+1,j'}$ and $x_{i,j} \leq x_{i+1,j'} + w_{i+1,j'}$ which means that the right side of $R_{i,j}$ lies right of the left side of $R_{i+1,j'}$.

Finally, since we minimize the sum of the $c$ values, as many as possible are set to 0. Each $c$ value set to zero corresponds to a realized adjacency, and thus the number or contacts in the resulting representation is maximized. □

## 7.4 Chapter Conclusion

In this chapter, we studied layout algorithms for semantic word clouds where the underlying graph is layered. We found that we could compute an area minimal layout in polynomial time, and could maximize contacts for two layers in linear time. Lastly, we introduced an ILP formulation to solve the problem for an arbitrary number of layers. The complexity of the problem for $L \geq 3$ layers remains open.

Our problem formulation differs from that of the CROWN problem on one significant point, namely, rectangles representing non-adjacent vertices should not share a contact. This is a natural restriction for the design of semantic word clouds, and it could be interesting to understand it's impact for the currently known hardness and approximation results.

CHAPTER 8

# Interactive Semantic Word Clouds

Semantic word clouds offer an interesting range of theoretical challenges, but as we have shown in Chapter 1 and 7, the most interesting problems are NP-complete. This implies that there are no efficient exact algorithms to exploit for the human-in-the-loop approach. To remedy this, we present in this chapter an interactive semantic word cloud layout tool, the leaves the optimization tasks in the hands of the user. To facilitate the user's task, a high-quality layout is presented, that can then be iterated upon and fine-tuned to obtain a final layout of high-quality that is also aesthetically pleasing according to the user's personal criteria. This chapter is based on joint work with Michael Huber and Martin Nöllenburg, and was presented at PacificVis 2023.

The first layout algorithm for semantic word clouds was presented by Cui et al. [CWL+10], and their technique motivated further work into efficient systems to create semantic word cloud visualizations. While there are several algorithms to generate such word clouds [BKP14], there is currently no interactive tool that allows the user to fine-tune these semantic word clouds in a human-in-the-loop way. The online tool implemented by Barth et al. [BKP14] lets a user generate layouts using many different algorithms and allows for some limited dragging operations on the words, as well as deletion but neither operation is supported by algorithms that update the layout. The number of relevant word pairs in a typical data set far exceeds the number of word pairs that can be adjacently placed by any planar word neighborhood structure. Fully automated solutions must therefore favor some of the many semantic links to be represented at the cost of missing others, purely based on numeric similarity scores. But only a human user, having deeper semantic knowledge about the underlying text, knows which of the chosen pairs are most relevant, which ones are missed, and which ones are less important. Hence, human-in-the-loop fine-tuning can lead to more user satisfaction and semantic accuracy of the word clouds as meaningful text summary visualizations. While a tool such as EdWordle [WCB+18] allows the user to easily edit a word cloud, having full manual control can be overwhelming for humans, and one should rather aim to combine the

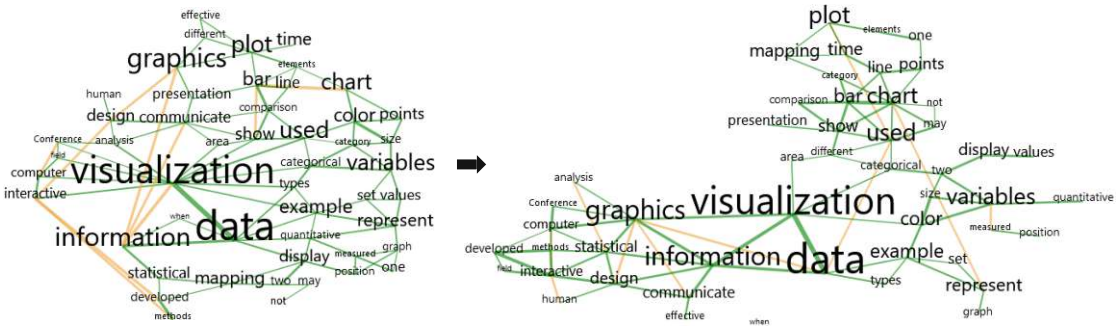



Figure 8.1: Semantic word cloud initially generated by our system from the "data and information visualization" Wikipedia page on the left. Realized adjacencies are shown in green and the main missing adjacencies in yellow. On the right the same word cloud after local user improvements. More adjacencies are realized, remaining large missing edges are shorter, and we can see three clusters emerge.

computational power of automated algorithmic solutions with the expert knowledge of the user. While the authors of EdWordle argue that a semantic word cloud could be used as input to their system to be edited, there is currently no simple way of importing a semantic word cloud into their tool. Additionally, any updates on the layout are focused on preserving existing neighborhoods and the compactness of the layout. Since the algorithm has no knowledge of the semantic relationships of the individual words in the word cloud, a single displaced word could completely perturb the existing neighborhoods, leading to a flawed layout, where the user has no guarantee or feedback about its semantic quality.

To address this gap, we introduce MySemCloud, an interactive human-in-the-loop semantic word cloud editor. Our tool generates a semantic layout of high quality using an algorithm inspired by Cui et al. [CWL+10], that performs well on the relevant evaluation criteria for word clouds [BKP14]. The main contributions of MySemCloud concern the subsequent editing steps and are two-fold. Firstly, we propose *metric guides* which allow the user to visualize the underlying semantic relationships within the word cloud, e.g., via the links between related words as shown in Figure 8.1, as well as guide the user towards potential improvements of the layout. Secondly, we propose *semantic-enhanced* interactions. While most interactive systems focus on changing the appearance of the words, our system is focused on preserving the semantic quality of the word cloud. Accordingly, MySemCloud includes a smart dragging tool that considers the neighbors of dragged words and updates their position while preserving the visual stability of the layout. Additionally, extending EdWordle's compactifying layout updates, our system is able to consider the semantic relationships when updating the layout, which allows us to focus on maintaining the most meaningful adjacencies.

**Structure of the chapter:** We first present an overview of the word cloud literature and current state-of-the-art layout algorithms (Section 8.1). Then, in Section 8.2 we

describe our layout algorithms, as well as the quality metrics we are designing our system for. In Section 8.3 we introduce the semantic-aware display as well a our semantic-enhanced interactions. Lastly, in Section 8.4, we present an user study to evaluate our system. MySemCloud can be accessed on `ac.tuwien.ac.at/mysemcloud`, where the code has been made available as well.

## 8.1 State of the Art

Word clouds were introduced in the early 2000's [VW08], but were known then as tag clouds. In tag clouds, the words that occur the most in an input text are scaled proportionally to the frequency at which they appear in the data set and displayed in lines, laid out alphabetically or by descending frequency. To improve on these layouts, more compact display methods were proposed [SKK+08] that focused on packing the words more tightly. These compact tag clouds were quite similar to visualizations created by designers by hands, as for example *pile of words* graphics that appeared in 2008 in the Boston Globe [Sch08]. Word clouds were further popularized by the Wordles website [VWF09], that allowed users to generate their own tag clouds, where the words are colored, displayed in a compact setting and sometimes rotated vertically. Appealing word clouds are often more than just compact layouts. For example, in Shapewordles [WCZ+20] the user chooses a (potentially complex) shape to draw the word cloud in. Extensions to the traditional word cloud design also include maps, where words are displayed on geographically significant areas [BCL+16, BGL+21, LDY18].

### 8.1.1 Semantic Word Clouds

The concept of using word placement in the plane is a logical way to encode more information within a word cloud. With semantic word clouds, the spatial distance between two words carries semantic meaning, namely placing closely related words nearby each other. While there exist different methods to compute a word similarity matrix [SSW+17, CWL+10, WPW+11], our focus lies in the layout algorithms. Cui et al. [CWL+10] proposed one of the first methods to generate such a semantic layout. Using multi-dimensional scaling (MDS), they computed 2D-coordinates for the words that approximate the desired relative distances to the other words. This technique usually results in a sparse layout. To avoid white space, they construct a Delaunay triangulation of the layout, on which they use a system of attractive forces preserving the neighborhoods. A similar result can also be achieved using a system of forces not on a triangulation of the layout but on the similarity graph itself [XTL16]. Wu et al. [WPW+11] proposed an alternative to the force-based compaction by using seam carving. They identify vertical or horizontal sections of the drawing that are empty and remove them from the drawing. The semantic word cloud layout problem has also inspired research with a more theoretical focus. When representing the words by their rectangular bounding boxes, it is possible to transform the semantic word cloud problem to one reminiscent of rectangle contact graphs. In contact graphs, the edges of an underlying graph are meant

to be realized by a proper edge contact between two boxes representing their respective vertices. In the Contact Representation of Word Networks (CROWN) problem, the number of edges of the input similarity graph that are realized in the contact graph has to be maximized. Barth et al. [BFK+14] proposed approximation algorithms to solve the problem on restricted graph classes which were later improved by Bekos et al. [BvDF+17]. Barth et al. [BKP14] compared multiple semantic word cloud algorithms using the most common semantic word cloud evaluation metrics. To our knowledge, there is currently no semantic word cloud layout system that offers an interactive component. This is a natural extension of the model when considering the amount of interest that interactive word clouds have generated [JLS15, KLKS10, WCB+18]. The dynamic semantic word clouds introduced by Cui et al. [CWL+10] have been studied further. They explored word clouds generated from a collection of documents at different timepoints. Not only was semantic relatedness encoded with proximity, but placement was also used to accommodate later changes of the data, e.g., words needing space to grow between two timestamps. Binucci et al. [BDS16] designed a layout algorithm that creates such word clouds over time without a-priori knowledge of the complete data collection.

### 8.1.2 Interactive Word Clouds

Visualization construction and authoring tools are of great interest to the information visualization community. Thus, creating interactive tools was a natural next step to the growing popularity of static word cloud layout systems. One of the first tools created was ManiWordle [KLKS10] which allowed the user to change the font, the color and orientation of the words as well as their position to potentially modify the whole layout. The interactive system allows users to fine-tune an automatically generated layout to match their personal aesthetic criteria. To maintain a compact layout, ManiWordle recomputes a layout using the Wordle algorithm but only considering the unedited words. In WordlePlus, Jo et al. [JLS15] extended these interactive environments to multitouch systems. To update the word cloud after the changes, boundary words are moved in the gaps left by a potential edit. While the interactive component is positively received, updating the word cloud itself remains a challenge as both solutions tend to disturb the mental map of the user significantly. Wang et al. [WCB+18] proposed EdWordle, a solution using rigid body dynamics to preserve the neighborhoods of the unedited word as well as to preserve stability of the layout. While the authors of EdWordle argue that their tool allows users to edit semantic word clouds without destroying the layout, there is currently no straightforward method of generating an initial semantic layout with EdWordle, and additionally the user has no ability to conserve the semantic quality during the edits. If a user moves a word, its direct neighborhood is lost, and the user has no knowledge of whether or not significant semantic information was lost. Similarly, while the remaining neighborhood is preserved, there is no guarantee of its actual semantic quality. The visualizations that were created by designers using EdWordle are reminiscent of the semantic word cloud thematic clustering, which highlights their strength as an information communication tool, but to create those word clouds from scratch is time consuming and can be overwhelming. Also, if the user does not have expert knowledge

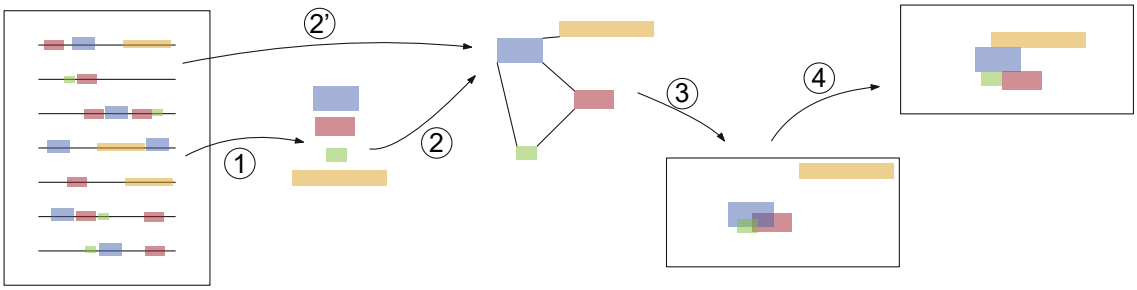



Figure 8.2: Steps to create an initial semantic word cloud layout: (1) stop words are removed, the remaining words are stemmed, and the top $k$ words are selected, (2) from those words and (2') the similarities computed from the co-occurrences in the input text (3) the words are laid out in the plane using MDS, and (4) attractive and repulsive forces are activated to obtain a compact, but overlap-free initial semantic word cloud layout.

of the text, they might not have the information necessary to create such a layout. Thus there is a need for a tool which provides a good quality semantic layout as a starting point that a user can easily fine tune subsequently, as well as information about the semantic similarities of the words in the text and interactions tailored towards preserving the semantic relationships in the layout.

## 8.2 Semantic Word Cloud layout

The general problem of generating semantic word clouds has been studied in depth by Barth et al. [BKP14]. Here we present our method in detail, which has been inspired by several works and tailored to our needs and implementation choices (JavaScript and D3.js). There are two main steps to the generation of a semantic word cloud layout. The first step involves generating from an input text a similarity graph and the second step concerns the actual layout of the words in the plane. An overview of our system can be seen in Figure 8.2. The goal is to generate a layout, where the relative position of two words indicates their similarity or lack thereof. There exist multiple metrics to describe the relationships between the words in the input text, and in the visualization, word pairs that score highly on the chosen relatedness metric should appear closer together than word pairs with a low score. This results in thematic clusters in the final layout, where the user is then able to understand the content and how different terms are related in the text source.

### 8.2.1 Constructing the Semantic Similarity Graph

The first step is to extract the words from the given text, which will later form the vertex set of our similarity graph. Using the natural language processing (NLP) library Natural [UEM11], we first remove irrelevant words from the text, for example “that”, “the”, “for”, ... that are not significant and should not be displayed. We then use stemming on the remaining words, meaning words are shortened to their meaningful

stem, e.g., the words "explanation" and "explaining" would both be understood by the algorithm as their stem "expla". For each of these stems we choose one of the words of the text as the representative of all the words with the same stem. This allows us then to accurately rank word frequencies, and to choose the top $k$ words that are found (see step (1) in Figure 8.2). We found that in most cases $k = 50$ is sufficient to cover the main themes of the text without overloading the visualization. This step is already sufficient to create a word cloud without additional semantic information. Finally for each chosen word, the number of times the stem occurs is summed up and we scale the font size of the word proportionally to this frequency.

To evaluate word similarities, further pre-processing is required. For a pair of words $w_1, w_2$, their similarity is calculated using the Jaccard similarity, which performs slightly better than the cosine similarity [BKP14], a common alternative. We calculate it in the following way. Let $S(w)$ be the set of sentences the word $w$ appears in. Then the Jaccard similarity of two words $w_1, w_2$ is a score in $[0, 1]$ given by:

$$s(w_1, w_2) = \frac{|S(w_1) \cap S(w_2)|}{|S(w_1) \cup S(w_2)|}.$$

From this we create a complete weighted graph $G = (V, E)$, where the vertices in $V$ represent the top $k$ words selected and the edges are weighted by the Jaccard similarity $s(w_1, w_2)$ for each edge $(w_1, w_2) \in E$. Some word pairs might have little or no similarity, meaning the two corresponding words rarely occur in the same sentence; thus we remove edges with Jaccard similarity below some threshold $\sigma$ from the graph. We set $\sigma = 0$ as a default, removing only edges corresponding to words that never co-occur (see step (2) in Figure 8.2).

### 8.2.2 Creating the Initial Layout

Using multidimensional scaling (MDS) [GKN04], we initialize the positions of the vertices of $G$ in the plane. At this step some words may be tightly clustered together and overlap, while others may be spread much further apart (see the example layout of step (3) in Figure 8.2). We then apply a forced-based system to the graph to adjust word positions, meaning we assign forces to the edges and vertices of the graph and then use these forces to simulate the motion of the vertices. It is important that words do not overlap one another so we must consider node overlap removal methods [CPPS20] that rely on a force system to naturally extend our layout algorithm. We calculate the distance between each pair of words in the x and the y dimension (see Figure 8.3a). If their bounding boxes overlap, the words are either pushed vertically or horizontally away from one another, in the direction that resolves the overlap fastest. That is, if there is more overlap in the x-dimension, the force will be applied vertically, thus ensuring a top/bottom or side contact between the two boxes remains after the resolution of the overlap (see step (4) in Figure 8.2). This method is similar to the Force-Transfer-Algorithm introduced by Huang et al. [HLSG07].

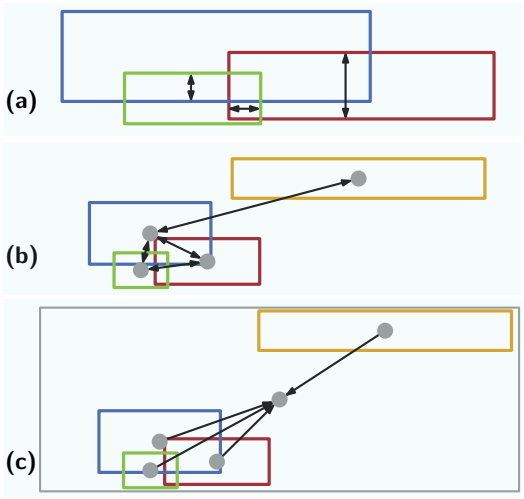



Figure 8.3: The force system to create the initial layout, **(a)** shows the repulsive forces that push overlapping words away from one another to prevent the bounding boxes from overlapping, **(b)** the attractive forces that pull all words toward their similar neighbors, proportionally to their respective similarity score and **(c)** the attractive forces that pull all words towards the center,

To obtain a compact layout, we must also apply attractive forces between each pair of words that is connected by an edge in $G$ as well as a centering force on every word. Consider a word $w_1$ that shares an edge with a word $w_2$. Then there is a force from the center point of the bounding box of $w_1$ oriented towards the center point of $w_2$ that is scaled by the value $s(w_1, w_2)$, meaning more strongly related words have a stronger attractive force (see Figure 8.3b). To ensure that our layout is displayed in the center of the canvas, we also add a weak attractive force from each word to the center of the canvas (see Figure 8.3c). Since our graph is dense, the force system might struggle to find a stable layout immediately, thus, we let it recompute iteratively new positions while decreasing the strength of the forces before we stop its computation and we obtain a final layout. Since our system is interactive, it is undesirable to let the system stabilise itself for too long as it affects the responsiveness of our system negatively. But it is also necessary to not stop it too early either to ensure our layout is of sufficiently good quality. We found experimentally that 1000 iterations, which could be computed in about 230ms, are a good compromise.

### 8.2.3 Semantic Word Cloud Quality Metrics

To design an automatic word cloud system, we often rely on optimizing quantitative metrics that measure the aesthetic qualities that are desirable in such a visualization. For classical, non-semantic word clouds, creating a compact design is the main visual criterion, and it remains an important aspect for semantic word clouds, too. But we must also consider metrics for the semantic quality of the visualization. Next, we

introduce the main quality metrics that are relevant to evaluate semantic word cloud layouts [BFK+14, BKP14, BvDF+17].

The semantic quality can be measured in two main ways, the first being *realized adjacencies* [BFK+14, BKP14, WCB+18]. When modeling the words as their rectangular bounding boxes, a realized adjacency corresponds to a segment contact between two boxes that share an edge in the semantic graph, see the highlighted edges in Figure 8.4. As this edge is weighted, the metric is weighted as well, meaning that it is better to realize a contact between two highly related words over one or more contacts of low weight. This metric captures the notion that the simplest way to understand that two words are related is if they are directly next to one another. As the boxes themselves are an abstraction of the words, rather than checking for a proper contact between two boxes, we check if the bounding boxes of two words $w_1$ and $w_2$ overlap. More precisely, assume $w_2$ has the smaller bounding box. We artificially inflate the size of its bounding box by 20% and check if it overlaps with the bounding box of $w_1$. Experimentally, we found that limiting a contact to a distance of 0 between two bounding boxes was too strict and words that visually appeared to be in contact were not counted as not realized. For a semantic input graph $G = (V, E)$ and a word cloud layout $\Gamma$ of $G$, the value $r(\Gamma) \in [0, 1]$ of the realized adjacencies $E' \subseteq E$ according to the above definition is calculated in the following way:

$$r(\Gamma) = \frac{\sum_{e \in E'} s(e)}{\sum_{e \in E} s(e)},$$

where $s(e)$ is the similarity score (weight) of the edge $e$ in $G$. To realize every adjacency, the input graph would need to be planar [BGPV08]. But semantic similarity graphs are dense, almost complete graphs, so this value $r(\Gamma)$ is often low. Nevertheless it is still an effective method to compare two layouts as that value can easily double from one drawing to the other. For an arbitrary graph $G$, finding the maximum realizable adjacency value is an NP-hard problem [BFK+14]. It was found that the cycle cover algorithm [BvDF+17] has the best performance for this metric [BKP14].

The second semantic quality metric is *distortion* [BKP14], which compares the distance of each word pair to their similarity score. Distortion can be seen as a relaxation of realized adjacencies as two highly correlated words do not need to touch but can instead be sufficiently close to indicate a meaningful relationship in the visualization, as shown by the colored edges in Figure 8.4. It also reflects the notion that unrelated words should not be close to one another, which the realized adjacencies metric fails to properly account for, since in its commonly accepted definition there is no penalty when two unrelated words touch. But with distortion, if their similarity value is low then they should be far away from one another in the plane. The distortion value $d(\Gamma)$ of a layout $\Gamma$ of $G$ is computed using Pearson's correlation coefficient $\delta(\Gamma)$ between the (dis)similarity matrix and the distances realized in the plane:

$$\delta(\Gamma) = 1 - \frac{\sum_{(u,v) \in E} (1 - s(u,v) - \overline{(1-s)})(d_\Gamma(u,v) - \overline{d_\Gamma})}{\sqrt{\sum_{(u,v) \in E} (1 - s(u,v) - \overline{(1-s)})^2 (d_\Gamma(u,v) - \overline{d_\Gamma})^2}},$$

where $1 - s((u, v))$ corresponds to the dissimilarity of $u$ and $v$, $\overline{(1-s)}$ is the average dissimilarity value in $G$ and similarly, $d_\Gamma(u, v)$ is the minimum distance between the bounding boxes of $u, v$ in $\Gamma$ and $\overline{d_\Gamma}$ is the average distance in $\Gamma$. The distortion is then defined as

$$d(\Gamma) = \frac{\delta(\Gamma) + 1}{2},$$

as $\delta(\Gamma)$ has its values in $[-1, 1]$. A value of $d(\Gamma) = 1$ indicates that every word is positioned at an ideal distance from any other word, 0 indicates the inverse and a value of 0.5 signals that there is no correlation between similarities and distances. Barth et al. [BKP14], who first introduced the distortion metric, found that the seam-carving algorithm [WPW+11] performed well with this metric.

Both metrics help us gauge the semantic quality of a layout. To evaluate its overall aesthetic quality, one can further consider *compactness* [BKP14, WCB+18]. The compactness $c(\Gamma) \in [0, 1]$ of a layout $\Gamma$ of $G$ represents the ratio of the space used by the words over the total space available in the bounding box of $\Gamma$. More precisely,

$$c(\Gamma) = \frac{\sum_{v \in V} a(v)}{a(\Gamma)},$$

where $a(v)$ represents the area of the bounding box of $v$ and $a(\Gamma)$ is the area of the bounding box of the entire word cloud. Most non-semantic word cloud layout methods achieve high values of compactness as they can form a tight packing without considering relative word placement. When semantics are considered, the need to separate some words from each other to obtain good distortion values often leads to lower compactness values.

There exist further metrics [BKP14] to evaluate the visual quality of a word cloud layout, namely *uniform area utilization*, which requires the words to be evenly distributed over the canvas. One can also consider the *aspect ratio* of the layout, where a ratio of 1 could be desirable, or one closer to traditional media formats like 16:9, in contrast to extreme aspect ratios, which might make the visualization difficult to read.

As our main interest point is the semantic quality of the layout and how it can be maintained during interactive steps, we will focus our attention on the realized adjacencies and distortion values of our layout, but retain the compactness metric in our quantitative evaluation as it is the most established of the three aesthetic layout quality metrics.

## 8.3 MySemCloud

MySemCloud is designed as a tool for a general public audience, with no deeper design expertise. It is created for users who wish to summarize familiar texts with informative word clouds in different media forms (presentation, social media) and focus on the information delivery. The target user is expected to have expert knowledge of the input

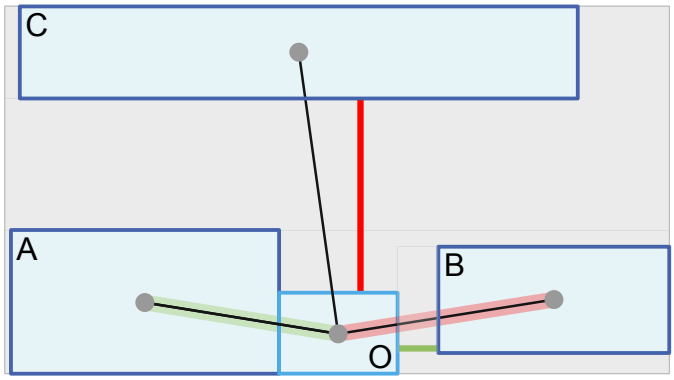



Figure 8.4: Word O is strongly connected to every other word. It successfully realizes its adjacency with word A (green highlight), but not with the word B (red highlight) as the boxes do not touch, but the distortion value with word B is good (green edge) as they are still close, unlike the distortion value with word C (red edge). The compactness corresponds to the ratio of the blue area over the area of the gray bounding box.

text, but limited expertise in graphic design. MySemCloud should be simple to use to ensure the user can quickly achieve a desired result. Therefore we focused on repositioning operations that are directly made on the visualization canvas. It is meant to be an alternative to existing editing tools which focus heavily on aesthetics and extensive design expertise.

In this section we present the technical details of our novel interactive semantic word cloud editing tool MySemCloud. Typically, a word cloud for a given text computed using the approach outlined in Section 8.2 is of good quality but does not take subjective user preferences into account yet. Since current algorithmic methods cannot predict which semantic relationships a user prefers to highlight over others, MySemCloud implements intelligent interaction modes and visual aids meant to guide the user in fine-tuning the computed word cloud themselves.

To help the user during their desired editing steps, MySemCloud provides two smart support tools, a *semantic-aware* display and *semantic-enhanced* interactions. All of these tools can be toggled on via the user interface Fig. 8.5.

### 8.3.1 Semantic-aware Display

The *metric guides* are a system of display layers meant to translate the underlying semantic data into visual cues that help the user understand and edit the layout while optimizing its visual and semantic quality. Specifically they correspond to three options that each can be toggled on or off together or separately and change the user interface.

**The adjacencies metric guide** lets the user display the edges of the semantic word similarity graph. When toggled on, for every edge in the semantic graph whose endpoints

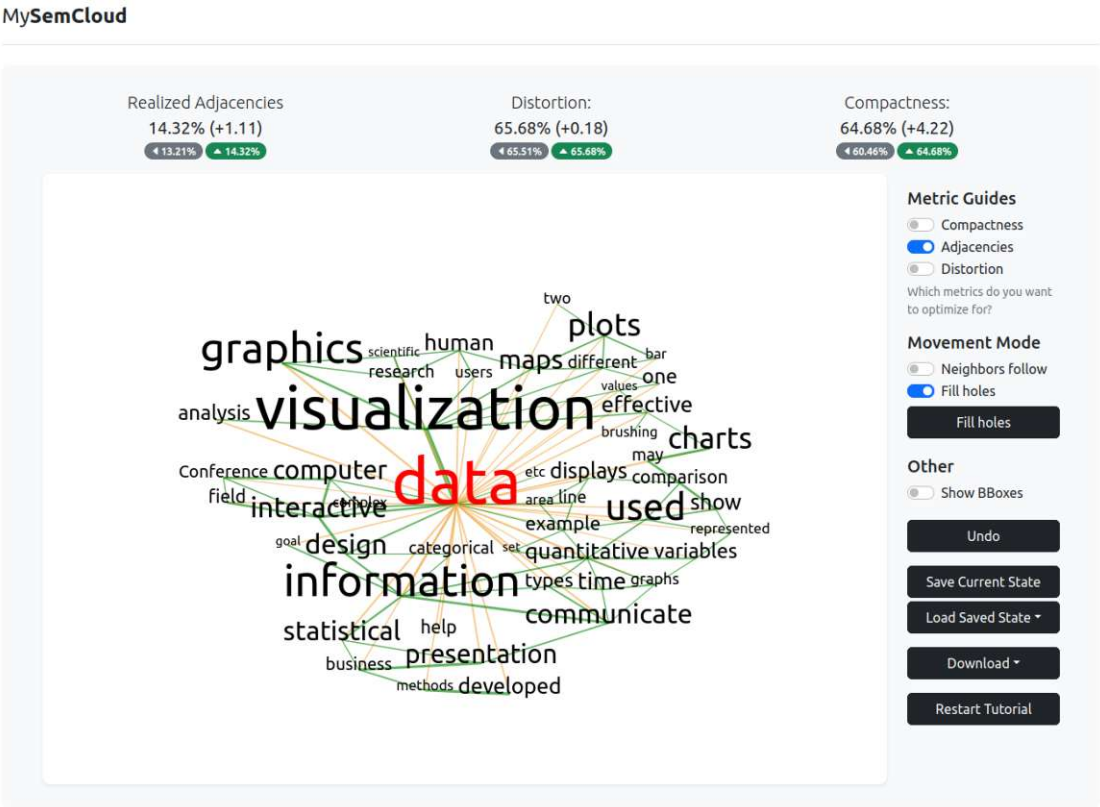



Figure 8.5: The user interface of MySemCloud, with the information visualization Wikipedia page as an input dataset.

are reasonably close to each other in the visualization to be considered an adjacency (see Section 8.2.3), the edges are displayed using a green segment between the center points of the two words corresponding to the edge's endpoints. The width of the segment is scaled proportionally to the similarity value of the edge. Additionally, some missing adjacencies are also shown in yellow. Since the graph is very dense, we only focus on showing the most significant missing contacts in the graph. We sort the list of missing adjacencies and select the ten highest-weight edges that are not realized, as shown in Figure 8.6a. This helps the user see where strongly connected components lie as well as the main missing adjacencies.

To ensure that all the data is visible, we offer a secondary view: when selecting a specific word using the right click, we replace the display of the main missing adjacencies of the cloud, with a display of all the edges that are incident to the selected word as shown in Figure 8.7. When the adjacency metric guide is active, if a user wants to move a word, they have the information of the word's adjacencies in its starting position, adjacencies that will likely be lost, and the adjacencies that might be realized in its new neighborhood, and thus make a decision of how to best adjust the position of a word. The user might

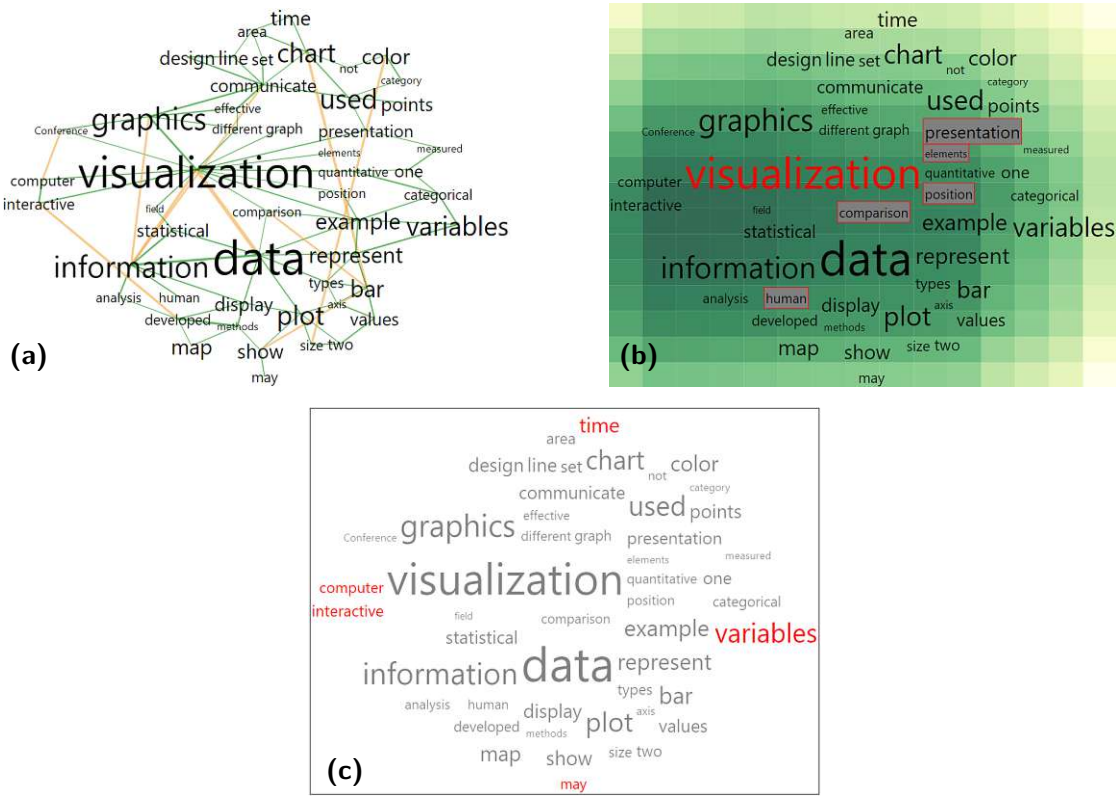



Figure 8.6: The different metric guides, **(a)** shows the realized adjacencies guide with the realized edges in green and the strongest missed adjacencies in yellow, **(b)** shows the heat map that indicates the positions with the highest distortion values for the selected word “visualization" in red and highlights the five most misplaced words, and **(c)** shows the compactness metric guide with the words stretching the bounding box highlighted.

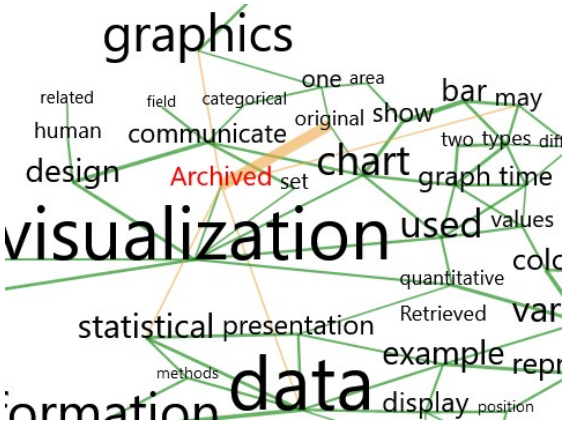



Figure 8.7: The right click operation under the adjacencies metric guide show the five links of the selected word “Archived”. We see one realized adjacency with “visualization” and five missing ones.

also decide which missing adjacencies are globally too important not to be realized and quickly identify those in the general view.

**The distortion metric guide** shows to the user which positions are, or are not, semantically meaningful. When selecting a word with the right click in the distortion view, a heat map will be displayed, where a darker green shade indicates positions that achieve high values for the distortion metric for the chosen word, and pale yellow shades represent positions that realize low values for the same metric. To create the heat map, we produce a tiling of the bounding box that the word cloud currently occupies. We compute a new distortion value by updating the length of the edges incident to the selected vertex only, then use a color gradient to associate a shade with the values obtained on a scale from dark green to pale yellow. This creates a color scale that indicates in which areas of the cloud the strongest neighbors of the selected word lie, and in which area the unrelated words are. The resulting view can be seen in Figure 8.6b. When turning the metric guide on, the words with the most negative impact on the distortion are highlighted in grey. To compute this, we calculate for each edge $e$ of the input graph an *ideal length* $\ell(e) = (1 - s(e))\frac{D}{2}$, where $D$ is the longest distance between two words in the visualization. We calculate the *penalty*, i.e. the difference between the ideal length and the actual length of $e$. If two words are unrelated and the difference is positive, they are too close and the penalty is squared. For each word we sum the penalties incurred with all other words, and finally highlight in grey the five words that achieve the highest sum.

**Lastly, the compactness metric guide** helps the user create a more space-efficient layout. When active, the bounding box of the word cloud is displayed, and the words that are on the boundary are highlighted (see Figure 8.6c). A user interested in creating a more compact layout can then select a boundary word and, using any of the two semantic metric guides, find a new suitable position that results in a more compact layout. This guide can be used alone, but as it only optimizes towards compactness, it is more relevant for semantic word clouds when used alongside the distortion or adjacencies guide. When used with a semantic guide, the user can more easily consider the global appearance of the word cloud while improving its semantic quality.

The three different views can be used individually or in any combination. Thus, the user can choose how to edit the layout in a way that can preserve the important neighborhoods, or they can choose to improve it by using the information from the underlying data set. While some views could potentially contain more information, we chose to prioritize simpler views to encourage the user to layer views on top of one another.

### 8.3.2 Semantic-enhanced Interaction

The default interaction step of MySemCloud is a drag-and-drop operation, where the user moves a word from one place to another. Once the new word is in place, we resolve any overlaps that were induced by the move. These edits cause minimal disturbances of the user's mental map and are useful for precise refinements of the layout. We additionally

provide two semantic-enhanced edit modes. Updating a force-based layout is often a challenge: a single local move can greatly perturb a graph's layout. Thus, our semantic-enhanced interactions are not only focused on incorporating the semantic graph with the interaction, but also on maintaining the stability of the previous layout. After any move is done by the user, the values of the quality metrics of the layout are updated to give the user direct feedback about how much impact the move they made has had.

**The *neighborhood-follows* mode** helps preserve the distortion and adjacencies in the graph. Specifically, when this mode is toggled on and a word is moved, the positions of its strong neighbors will be updated as well. There are two main components to this step.

The first component concerns the selection of the vertices whose position should be updated. We compute a breadth-first search tree of $G$ rooted at the moved word $w_1$, and add vertices from that tree to the *relevant vertices list* in the following way. We first consider all the children of $w_1$. If those have similarity value with $w_1$ higher than our threshold $\theta$, they are added to the list. We then look at the children of those selected vertices. We compute the ratio of their similarity value with their parent over their depth in the tree. If those are higher than $\theta$, they are similarly added. We continue down the tree using the same ratio of similarity with the parent divided by depth of the vertex until all vertices have been considered. This method ensures that we are less likely to select grandchildren of $w_1$, and will do so only if they are strongly related to their parent (and that parent is strongly related to $w_1$). We set $\theta = 0.1$.

The second component is the update of the layout itself. First the non-overlap forces and attractive forces are turned off. An *anchoring* force is added to every word, oriented from the center point of the word to its current position at the start of the move. For each edge linking a vertex of the relevant vertices list to their parent, we reactivate the attractive forces in the following way: all edges between $w_1$ and its children in the list are reactivated with their full strength, all the edges between the chosen children of $w_1$ and their own chosen children will be reactivated at partial strength, and every further edge will have its strength decreased proportionally to their depth in the BFS tree of $w_1$.

Finally, when the new positions are computed, the anchoring forces of the moved neighbors are updated to be directed to their current position, the overlap removal forces are reactivated and we compute the final layout. Using this move, when a word is dragged, its highly similar neighbors will follow it, thus preserving the important adjacencies in the graph. Since those following words might have strong adjacencies themselves, we search deeper in the tree to find if some are significant enough that further words might be moved as well. The anchoring force pulls any following words back towards their starting position. This helps to maintain the stability by not permitting the moved words to go too far. It also avoids significantly disturbing the distortion value of the layout as can be seen in Figure 8.8.

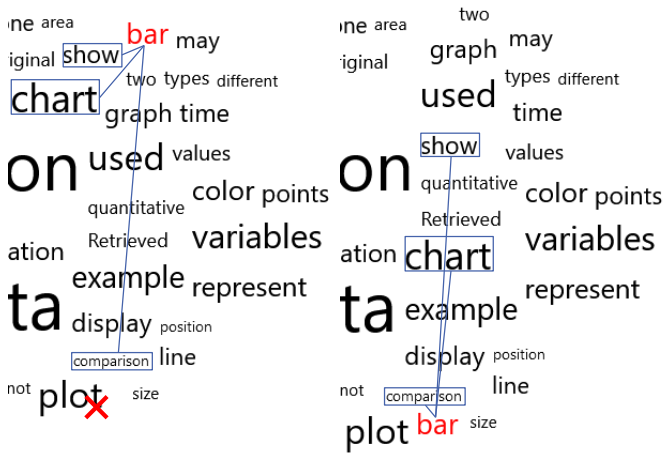



Figure 8.8: “Bar” is moved to the position marked by a red cross using the neighbors follow mode, the algorithm selects “show", “chart" and “comparison" as its significant neighbors. “Comparison" was close to the new position and is able to realize the contact after the move. “Chart" and “show" also move closer to “bar" but “show" has more similar neighbors in the upper part of the visualization and thus does not move too far away.

**The fill-holes mode** aims to preserve the compactness without damaging adjacencies. A common issue with interactive word clouds is that when a word is moved, the space left behind should be filled to re-establish the compactness of the layout. In MySemCloud, we resolve this issue by reactivating the forces of the system similarly to how we generated the first layout. Specifically all the attractive forces corresponding to the edges are reactivated, as are the centering and the non-overlap forces. This allows the layout to re-compactify itself, in a manner where words that are more strongly related to one another will more likely be pulled together into the hole created by a word move than weakly related words. This operation can also be triggered with a button, without needing a move.

When used together, these interactive modes tend to update the layout significantly such that a user’s mental image might be disturbed. The fill-holes mode tends to increase the value of realized adjacencies, as it attempts to close the gaps between words. The neighbors-follow mode can also be used for larger updates: when wanting to place a new topic in an entirely new area of the layout, it can effectively allow the user to move a cluster of related words at once.

MySemCloud further contains a mode to toggle the bounding boxes of the words, as it makes it easier to notice directly if the contacts are realized or not. This is also helpful with compactness as one can quickly spot gaps in the rectangle packing. It also contains an undo button which reverts the layout to its state before the last move was executed, as well as a button to save a certain state of the layout. The user can then load any

saved state at any point in the editing process, or recover an unsaved state using the undo button. Lastly, the values of the metrics are displayed for the current layout, the previous layout, as well as the best value achieved by a layout. After every move the values are updated and if the layout achieves a new optimum for any value, it is saved automatically.

## 8.4 Evaluation

We evaluate MySemCloud from two different perspectives. First, we explore in a controlled study how the semantic-aware display and the semantic-enhanced interactions are able to allow users to improve the quality of initial semantic layouts. Second, we present findings from a qualitative study during which participants were able to freely use MySemCloud to design word clouds of their own text data. We want to show that MySemCloud is not only able to generate word clouds of high semantic quality, but that it additionally is a good compromise between the existing interactive, but non-semantic word cloud editors and the non-interactive, semantic word cloud layout algorithms.

We implemented MySemCloud in JavaScript. The text submitted in the client is sent to a backend server running on Node.js that handles the semantic similarity computation using the NLP library *Natural* [UEM11] and generates the MDS layout. The final layout is computed in the client using the popular JavaScript visualization library *D3.js* [Bos] for the force layout computation and the rendering.

Both aspects of the evaluation were performed as a back-to-back in-person user study that took 45min during which each participant worked individually with the tutor. We recruited 20 participants (5 women, 15 men) who were students or researchers in Computer Science. None of the participants had used a word cloud creation tool or layout algorithm previously. One participant reported not being familiar with word clouds, and another reported having heard about semantic word clouds. All participants reported normal or corrected-to-normal vision, and had no color vision deficiencies. The study was conducted on a 27" LCD screen at a $2560 \times 1440$ resolution using a mouse as input device.

### 8.4.1 High-Quality Layout Creation

We introduced in Section 8.2.3 several metrics to evaluate the quality of a semantic word cloud layout. In the study, we focused on the two semantic quality metrics *realized adjacencies* and *distortion*, and on the non-semantic *compactness* metric. In this section we evaluate the quality of the layouts created by the 20 participants using MySemCloud. Specifically, we investigate how efficiently users can improve the quality of the layouts using the semantic-aware display and semantic-enhanced interactions provided in MySemCloud.

### Study Design

The study was conducted in three steps, an introduction and training phase, followed by a set of word cloud improvement tasks, and completed by a short questionnaire.

To start, the participants were given an introduction to semantic word clouds, as well as the definition and intuition behind each metric. They were then introduced to the tool, and each of its functionalities was explained. They were given time to learn to use the tool, and told what the tasks would consist of. During this training they could ask questions and familiarize themselves with MySemCloud.

**Data Sets.** The data sets used for the study were the following: (1) A summary of the book "Harry Potter and the Philosopher's Stone" [Moh] as a training data set and (2) the English Wikipedia page for the "European Union" [wik22] for the study tasks.

As the users of MySemCloud are expected to have some familiarity with the texts they are designing a cloud for, we chose text data sets covering topics of broader public interest. One participant reported no familiarity with the Harry Potter book, but all participants reported having sufficient knowledge of the European Union to understand the content of the layout being presented to them.

**Tasks.** Four tasks were given to the participants to evaluate the different aspects of our tool. They were asked to improve the value of a metric as much as possible within a given time. For task 1 they had to improve the *realized adjacency* metric, for task 2 the *distortion* metric, the *compactness* metric for task 3 and to improve in parallel as much as possible the values of the *realized adjacency* and *compactness* metrics for task 4. While the participants spent time training on the tool before the tasks were undertaken, we randomized tasks 1 through 3 to avoid systematic bias through leaning effects. We gave the participants 2 minutes for tasks 1, 2, and 3 and 2.5 minutes for task 4. We calculated the improvement rate achieved by each participant within each task for the targeted metric. We did not give participants a lot of time as we were interested at how efficient our design was at guiding the users during the edits. The last task was given more time as we expected it would be more complicated for the participants to optimize two potentially conflicting goals simultaneously. We focused on combining compactness and realized adjacencies as those have been the strongest focus of previous semantic word cloud layout algorithms [KLKS10, WCB+18, BKP14, BFK+14, BvDF+17]. The participants were not obliged to use the relevant metric views for each task or specific interaction modes, but were asked to choose the setup that they felt most efficient working with.

Once the participants had completed the tasks, they were given a questionnaire to describe their understanding of the quality metrics and to evaluate the difficulty of the task.

**Results/Findings**

Figure 8.9a shows the results of the four metric improvement tasks. Our hypothesis was that we believed candidates would successfully complete all four tasks, but struggle with task 4 when balancing the two different metrics. We found that the candidates were most successful with the compactness improvement task, as 19 participants improved the compactness of the bounding box. For the task of increasing the realized adjacencies values, the candidates were similarly very successful, with only two candidates failing to improve the layout. When the candidates were tasked of improving adjacencies and compactness in parallel, they were similarly successful. No participant failed to improve realized adjacencies on this second attempt, but some neglected the bounding box improvement.

Three candidates had chosen for the compactness task to completely ignore any semantic positioning by creating a tight packing of rectangles, but could not rely on this strategy on the combined task. Those participants successfully improved the layout when they had to preserve and improve its semantic quality.

The participants on average clearly outperformed the four automated layout algorithms on most tasks. The cycle cover algorithm, that is designed towards maximizing realized adjacencies was outperformed by 11 participants in task 1 as well as in task 4. No participant was able to obtain a better distortion value than what was achieved by the seam carving algorithm, but they usually performed better than most of the competing algorithms, confirming the anticipated advantages of our human-in-the-loop approach.

We also note that they were able to efficiently target the two metrics at the same time in task 4, unlike for example cycle-cover which is targeted towards adjacencies and performs poorly on compactness. Candidates in general did not heavily disregard one metric for the other as the best participants achieved good values for both metrics and similarly the worst performers tended to struggle with both metrics.

The task to improve distortion proved difficult, as eight participants did not succeed in improving the distortion value of the initial layout. During the interview, six participants reported that while they understood the intuition behind this metric, they had issues understanding how to translate it visually. When asked which metrics they thought were the most relevant for semantic word clouds, distortion was the best received metric with eight participants commenting that it was the most relevant metric for these layouts, two of those had reported having trouble with the distortion improvement task.

The difficulties with this task are likely due to the participants focusing on the larger words of the layout. Those, when moved, tended to heavily disrupt the initial layout, often lowering its overall quality. More successful participants focused instead on average and smaller sized words, which often were farther from ideal positions. Given additional targeted training and more time for the task, we suppose that candidates could have been more successful. An example of a word cloud with high realized adjacencies and distortion scores can be found in Figures 8.9b and 8.9c. Notice that high value distortion layouts are less densely packed, which is contrary to high realized adjacency and high compactness

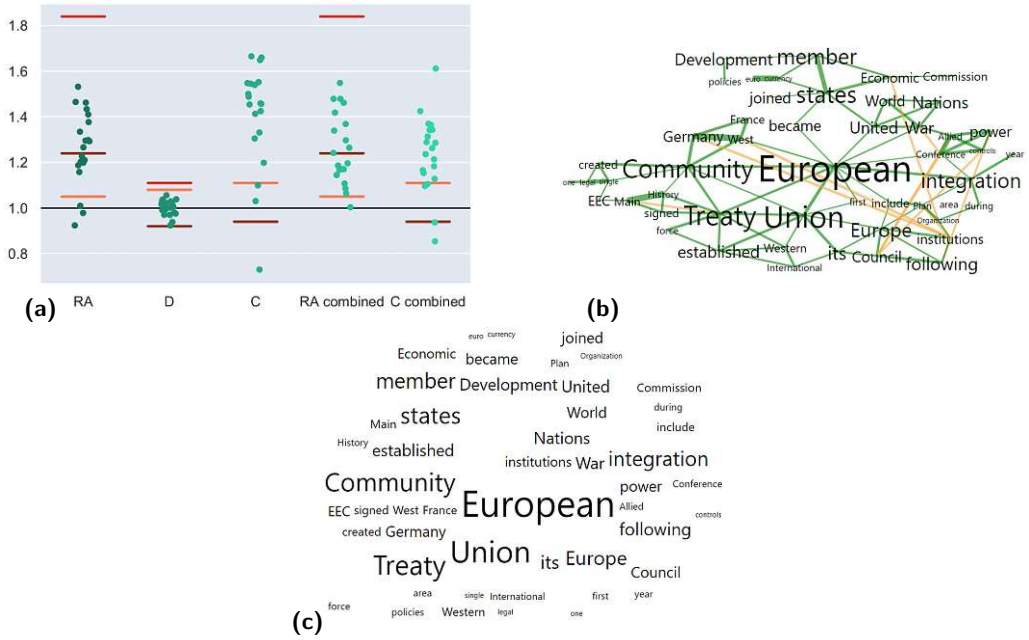



Figure 8.9: **(a)** Each point corresponds to the ratio of the metric value obtained by a participant for the task indicated by the column over the value of the starting layout. The left values in task 4 correspond to realized adjacencies and on the right is the compactness. The grey lines link values obtained in the same layout. A value of 1 means no improvement, anything above the line corresponds to a better value in the targeted metric and a value below 1 shows a worsening of the score. On average every task was successful, although distortion was the most difficult overall. The horizontal lines correspond to values achieved with the algorithms specified in the legend [BFK$^+$14, KLKS10, BKP14, WCB$^+$18]. **(b)** an example of a semantic word cloud achieving high realized adjacencies values and an improvement ratio of 1.84, **(c)** an example of a semantic word cloud achieving high distortion value corresponding to an improvement ratio of 1.11. Both layouts were created using MySemCloud in under 5 minutes and outperform the best automated layouts.

value layouts. As many participants naturally tried to augment the compactness of each layout even when the tasks did not require it, this can also explain the difficulties they encountered. One might consider an alternative definition to the metric, that is compatible with denser layouts, or that results in higher swings of the value of the metric which would give more feedback to the user regarding the efficiency of their edit.

We hypothesized candidates would understand the compactness metric most easily, as well as not have many difficulties with the realized adjacencies metric. Each participants graded the difficulty of improving each metric on a seven point Likert scale, where 1 meant the task was very difficult and 7 meant very simple. They found that compactness was the simplest, rating it a 6.3 (simple), distortion was the most difficult giving it a 3.95 rating (neither simple nor difficult), and they gave realized adjacencies a 4.65 grade

(slightly simple to simple).

### 8.4.2 Qualitative Evaluation

In this section, we want to understand how MySemCloud could be used as a word cloud design tool and how users approach it as a visualization tool to present a text of their choosing. We want to study the interest users have in semantic word cloud layouts and how much they value being able to edit and fine tune their layouts.

#### Study Design

For this task, we simulated a typical use case of our tool, where a user creates a semantic word cloud for a text they have a deep knowledge of. As our pool of participants contained 13 researchers and 7 students who had at least obtained a Bachelor's degree, each participant was able to choose a scientific text that they had expert knowledge of. Of the 20 participants, 10 chose a paper they were a main author of, 4 chose their thesis, and 6 chose a paper they had thoroughly studied.

A semantic word cloud was generated from the chosen input text using MySemCloud, and the participants were asked to edit the layout into a visualization of their liking. They were able to take as much time as they desired and could use any functionality of MySemCloud. Overall, the candidates took between 5 and 20 minutes to arrive at a final layout and had different design goals.

The candidates were then introduced to the semantic word cloud generation tool created by Barth et al. [BKP14]. The functionalities were explained and they were able to try out the different algorithms on the text they had previously chosen. They were also shown the layouts generated using the Cycle Cover algorithm [BFK+14], the layout computed using the Seam Carving method [WPW+11], the Inflate & Push layout [BKP14] as layouts which, respectively, achieved high values for the realized adjacency, distortion and compactness metrics.

They then completed a questionnaire covering their experience using MySemCloud, their impression of semantic word clouds in general and they were asked to compare MySemCloud to the non-interactive layout algorithms. Lastly, an interview was conducted during which each participant was asked about their design goals for their personal word cloud, their impression of the metrics and the quality of the visualization as well as their impression of MySemCloud.

#### Results/Findings

Our hypothesis was that participants would preserve the semantic grouping created by the original layout, and would focus on changing the placements of some words to more appropriate topic clusters.

**Design goals.** We identified three different types of design goals amongst the participants: the compact designs (13), the clustered designs (5) and the mind map designs (2).

An example of a compact design can be seen in Figure 8.10b. Here the participant did not edit the initial layout (Figure 8.10a) significantly, most of the changes are results of the interaction modes and overlap removal forces. The main aspects of those compact designs revolve around a strategic placement of the largest words toward the center. Our algorithm tends to draw the bigger words towards the center as they often have a very high degree in the similarity graph. This was rated positively by eight participants as it aligned with their design goal. The smaller words are naturally arranged on the periphery. Some of those words are moved, often using the hints given by the semantic metric guides, closer to the most related large neighbor. In the interviews, eight participants describe a layout with the main themes centered as an ideal layout. The resulting layouts appeared more compact, but due to their often rounded designs achieved on average a coverage of under 60% of the bounding box volume.

A related class are the clustered designs, e.g., see Figure 8.10d created from the layout of Figure 8.10c. Here we note that the final layout is less dense and multiple thematic clusters appear. Four participants preferred that the larger words were separated and serve as the centers of thematic clusters. In those word clouds, the larger words were spread out and the smaller words that were misplaced or at the periphery between two clusters were brought closer to a certain cluster. Every such design in our study achieved a compactness score of less than 50%.

The last designs are the mind map designs. In those cases, participants disregarded the initial layout and instead created from scratch a new layout, where small topic bubbles containing few words were spread around the canvas around the central most meaningful word.

We found that participants consistently spent time fixing the reading direction of some word pairings, e.g., the two words “induced" and “subgraph" should not be reverted. Additionally, they separated words that were loosely related when they had the same font size and appeared side to side, as they would otherwise appear visually as a word pair rather than two independent words.

**Metrics.** The importance of the metrics was evaluated next. The participants were asked which metric, if any, they were interested in when working with the MySemCloud layout. We hypothesized that participants would naturally lean towards compact layouts and realized adjacencies. Seven participants noted they mostly were interested in compactness. As for the semantic metrics, four reported paying attention to the distortion value, and five to realized adjacencies. They were then asked how the value of the metrics was correlated with the quality of the layout. Those answers did not align well with the participants’ personal design goals. Specifically, eight participants thought that distortion was the most meaningful metric, six thought distortion and realized adjacencies were

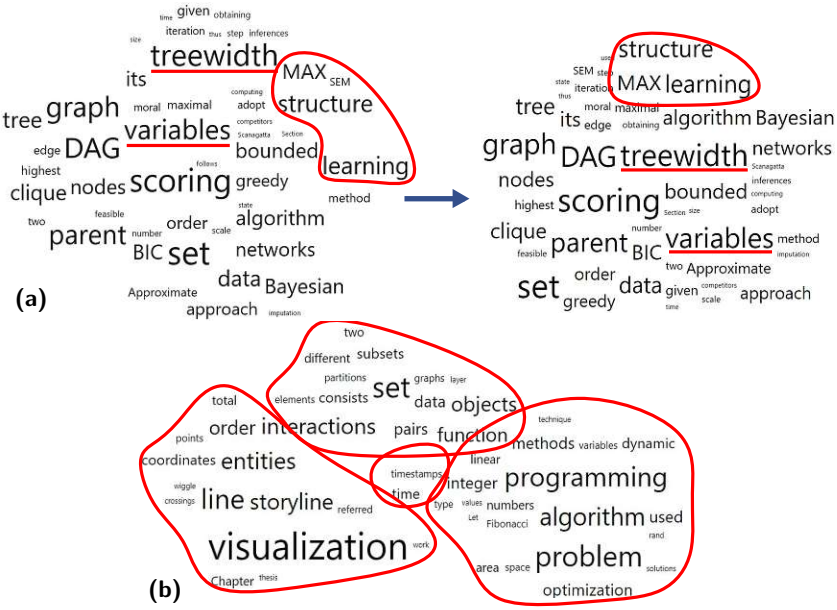



Figure 8.10: Example of semantic word clouds generated for task 2, **(a)**: a word cloud before and after the user fine-tuned the layout, the larger words have been organised in topic clusters and few smaller words have been moved, **(d)**: a word cloud created by a user, gaps separate the different themes which are themselves clusters together.

equally the most relevant and three thought that adjacencies were a better indicator of semantic quality. Compactness was not received as well as the general impression was that it was misleading. Five participants noted that higher compactness lead to worse layouts, and three that it was a secondary goal and only beneficial up to a point.

**MySemCloud.** We hypothesized that our users would enjoy the playful nature of the tool, and appreciate the novelty of semantic word clouds as opposed to the more commonly seen compact non semantic layouts. We also believed that the users would rate highly the ability to interact and improve the visualization over the current best performing layout algorithms. The impressions of MySemCloud as an interactive editor were very positive. Participants valued the simplicity and efficiency of the design (8). Nine participants highlighted the different interaction modes and six the metric guide views. Two participants were interested in the ability to see the underlying data and noted that data exploration was for them a strong use case for MySemCloud.

We evaluated our tool using the same questionnaire developed to evaluate ManiWordle [KLKS10] and EdWordle [WCB+18], as well an additional set of questions targeted at our system. We can see in Figure 8.11a that the design of my MySemCloud was found to be very efficient. We can note that creativity was not rated as highly by the users, which is to be expected as we designed the tool to create effective semantic text visualizations as a focus over aesthetics. Moreover, the semantic-enhanced interactions

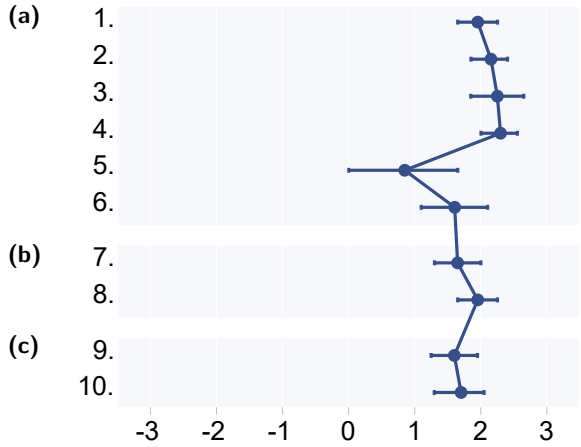



Figure 8.11: Participants used a seven point Likert scale to rate the following ten statements from *strongly disagree* (−3) to *strongly agree* (3): 1.) *It was easy to learn MySemCloud (MSC)*, 2.) *It was easy to use MSC*, 3.) *I liked to use MSC*, 4.) *It was fun to use MSC*, 5.) *I felt creative while using MSC*, 6.) *I am satisfied with the result*, 7.) *The metrics were understandable*, 8.) *The metric guides were understandable*, 9.) *MSC represented the data well*, and 10.) *MSC was faithful to the data*; **(a)** (1.–6.) covers the user experience, **(b)** (7.–8.) the metrics and **(c)** (9.–10.) shows that our tool was preferred to the automated layouts.

somewhat restrict the edits of the word clouds compared to the free drag-and-drop mode. The participants rated highly the quality of the visualization they created using the tool, and thought that the tool itself represented the underlying information of their chosen input text faithfully (see Figure 8.11c). Additionally, while some participants reported issue with the distortion metric, on average when considering all the metrics, the participants had a strong understanding of the optimization goals of the semantic word clouds (see Figure 8.11b).

Lastly, when comparing their experience of MySemCloud to the non-interactive layout algorithms, 16 participants preferred MySemCloud, with 7 indicating a strong preference. On their preference of semantic word clouds over traditional word cloud layouts, 17 participants preferred semantic layouts, two found that it depends on the use case and one participant found that semantic layout presented them with too much information. Additionally, two participants answered that they naturally assumed non-semantic word clouds had a semantically meaningful layout, and thus found them misleading.

### 8.4.3 Limitations

Interactive word clouds offer more than updating the position of words. For example ManiWordle [KLKS10] and EdWordle [WCB+18] allow the user to rotate or color words. Such functionality has not yet been implemented in MySemCloud, but could be appealing for users. The focus of MySemCloud so far is the semantic quality of the layout, but it

could be worthwhile to study how a system that supports both general aesthetics and semantics is perceived. Such an in-depth system could overwhelm the user, but as some participants in our study suggested, these are possible extensions of MySemCloud.

Additionally, we chose not to allow the user to interact with and modify the underlying data computed by the NLP algorithm, to ensure that it remains faithful to the input text. In the creative part of our user study, however, some participants had complaints about the limitation of the NLP library, e.g., as we often worked with text from mathematical publications, the word "theorem" could appear, which the participant considered to not be actually relevant, but the NLP algorithms considered it significant due to its high frequency in the text. Similarly, when dealing with technical vocabulary, stemming can fall short, e.g., the words "parameterized" and "parameter" being considered two independent words. Thus, the need to remove and add some words from the top-$k$ word list is a natural addition to MySemCloud. While editing the input data might help generate a better initial layout, users will still need to refine the visualization further. Therefore we chose to focus on the more difficult task of providing meaningful information to the user to guide the direct interactions with the layout, while adding data editing modes remains as future work.

With regards to the interactive modes, we noticed that the participants would sometimes move a word on the boundary slightly to trigger some compaction using fill holes; so having a button to trigger the compaction directly would be a natural addition. Lastly, the implementation of the bounding boxes sometimes caused visual confusion, and the participants would attempt to bring two words together that would not stay close. This is due to the perception of the bounding box by the user being different from the bounding box used in the algorithm. An implementation similar to EdWordle that considers the bounding box of each letter individually might lessen those issues.

## 8.5 Chapter Conclusion

In this chapter, we presented MySemCloud, a novel human-in-the-loop word cloud editor combining the strength of the semantic word cloud layout algorithms with those of interactive word cloud systems.

We found that users were often dissatisfied with layouts computed by state-of-the-art algorithms as they tended to focus on the wrong semantic relationships, had sometimes undesirable layouts for the largest words, and would misplace several words. The study participants, on average, outperformed the state-of-the-art algorithms on almost all quality metrics with our human-in-the-loop approach. We also found that the focus on compactness provided by previous word cloud editors was detrimental to a visualization of high semantic quality, and the lack of semantic information made the word clouds less interesting.

Overall, we showed that MySemCloud successfully bridges the gap between non-interactive semantic layouts and the existing non-semantic interactive tools. While some users did

not believe that they could successfully improve a layout and others were happy to spend a longer amount of time to completely recreate their layout, the large majority of our participants fell between those two extremes and engaged happily with our interactive system. We also found that our tool has a strong potential for data exploration, more so when the users do not have expert knowledge of the input data.

The neighbors-follow mode requires a certain similarity value as a threshold, and it, as well as the fill-holes mode, reactivate forces at a set strength coefficient. These parameters could be set by the user, and would offer flexibility at the cost of making the tool more complex.

It could also be interesting to study layout methods that take into account user preferences to generate the visualization. Some users are not willing to spend time carefully editing a layout, but still have preferences about which semantic elements should be highlighted. Such a user-centered layout algorithm would combine naturally with our human-in-the-loop fine-tuning system. Lastly, one could also consider extending this approach to handle time-varying data, as the words need to be laid out not only with regards to user preferences but also to enable morphing and optimize stability between subsequent word clouds.

CHAPTER 9

# Conclusion

In this thesis we presented some graph drawing problems, studied their complexity, and designed efficient algorithms for these problems. Our goal was to understand how to extend our own, and other known theoretical results, to practical applications. In Part I, we focused on node link diagrams and the vertex splitting operations for general graphs Chapter 3, bipartite graphs Chapter 5, in planar graphs to find outerplane drawings Chapter 4, as well as investigated efficient algorithms to reduce crossings in real world graph instances Chapter 6. In Part II, our focus shifted to contact graph representations with the semantic word cloud problem, where graphs are represented using contacts between rectangular boxes in the plane, where the rectangles have a fixed area and aspect ratio. In Chapter 7, we studied a setting where the underlying semantic word cloud graph is a layered graph, and in Chapter 8, we implemented an interactive user in the loop system to design semantic word cloud.

As is common in graph drawing, most problems that we encountered are NP-hard, which complicated the investigation into practical implementations of algorithmic solutions. We relied on a mix of heuristic methods, ILP formulations and exact algorithms to further understand and evaluate our findings, and introduce human-in-the-loop algorithmic considerations. In the next section, we want to consider the broader open questions relating to our main question, that is, how to efficiently combine expect human knowledge with exact algorithmic method to tackle complex tasks. For a detailed discussion on a specific problem, we refer the reader to the relevant chapter conclusion, where the main results are summarize, and the main open questions are listed.

## Outlook

The field of information visualization regularly inspires interesting graph drawing problems. Relying on heuristics is often sufficient for many applications, but, to understand a problem fully, it is important to know its complexity. Since many graph drawing problems are

NP-hard, creating practical algorithms to solve them is often impossible. Thus, in this thesis, we focused on combining human knowledge with algorithmic techniques to achieve more practical solutions to graph drawing problems. We believe that strongly adhering to the optimization goals dictated by the problem should be avoided. Not only is it a computationally hard task, but users also consider multiple aesthetic criteria at once in a manner that cannot be described by a function. Instead, one should focus on creating algorithms which allow the user to focus on their own personal quality criteria through fine-tuning.

We highlight the following tasks that we believe can lead to effective human-in-the-loop graph drawing algorithms. Computing an **initial layout** of high quality allows a user to create a mental map, an get a basic understanding of the dataset. The user can then identify areas in the layout that require changes, and one can propose many methods for **computer-aided layout fine-tuning**. This can be achieve through **computing quality metrics**, to guide the user towards certain choices that might improve the layout. For example it is simple to identify singular vertices responsible for a large amount of crossings, or placed far away from their neighbors. Additionally, this simplifies the algorithmic task. Rather than attempting to optimally compute a whole layout, if the targeted subproblem is restricted enough, exact algorithms might be achievable. For example, one can easily split a vertex to minimize its crossings, but a set of multiple vertices is computationally not tractable. Lastly, splitting a general task in a **pipeline of sub-tasks** limits the computational hurdles to manageable instances. In general, the process of creating and editing a layout can offer invaluable insight into the data being visualized, thus, any user wanting to create beautiful data visualizations within such a system can not only represent their data in a way that they believe would be most effective, but also understand the data itself better.

In this thesis, the practical solutions proposed for both settings very significantly, thus it would be interesting to pursue this investigation further, to identify a more defined series of tools necessary for such systems, or to see if, like graph drawing aesthetics, each problem invites very different solutions.

# Bibliography


[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[ABFS21] Faisal N. Abu-Khzam, Joseph R. Barr, Amin Fakhereldine, and Peter Shaw. A greedy heuristic for cluster editing with vertex splitting. In *Proc. 4th International Conference on Artificial Intelligence for Industries (AI4I)*, pages 38–41. IEEE, 2021.

[ADBF+15] Patrizio Angelini, Giuseppe Di Battista, Fabrizio Frati, Vít Jelínek, Jan Kratochvíl, Maurizio Patrignani, and Ignaz Rutter. Testing planarity of partially embedded graphs. *ACM Transactions on Algorithms*, 11(4):32:1–32:42, 2015.

[ADM+22] Lorenzo Angori, Walter Didimo, Fabrizio Montecchiani, Daniele Pagliuca, and Alessandra Tappini. Hybrid graph visualizations with chordlink: Algorithms, experiments, and applications. *IEEE Trans. Vis. and Comput. Graph.*, 28(2):1288–1300, 2022.

[AFK+12] Karin Arikushi, Radoslav Fulek, Balázs Keszegh, Filip Moric, and Csaba D. Tóth. Graphs that admit right angle crossing drawings. *Comput. Geom.*, 45(4):169–177, 2012.

[AFL08] Faisal N. Abu-Khzam, Henning Fernau, and Michael A. Langston. A bounded search tree algorithm for parameterized face cover. *Journal of Discrete Algorithms*, 6(4):541–552, 2008.

[AKK22] Reyan Ahmed, Stephen G. Kobourov, and Myroslav Kryven. An FPT algorithm for bipartite vertex splitting. In Patrizio Angelini and Reinhard von Hanxleden, editors, *Proc. 30th International Symposium on Graph Drawing and Network Visualization (GD'22)*, LNCS, pages 261–268. Springer, 2022.

[AKP+20] Alan Arroyo, Fabian Klute, Irene Parada, Raimund Seidel, Birgit Vogtenhuber, and Tilo Wiedera. Inserting one edge into a simple drawing is hard. In Isolde Adler and Haiko Müller, editors, *Proc. 46th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 12301 of *LNCS*, pages 325–338. Springer, 2020.

[AL04] Faisal N. Abu-Khzam and Michael A. Langston. A direct algorithm for the parameterized face cover problem. In Rod Downey, Michael Fellows, and Frank Dehne, editors, *Proc. 9th International Symposium on Parameterized and Exact Computation (IWPEC)*, LNCS, pages 213–222. Springer, 2004.

[AP20] Pankaj K. Agarwal and Jiangwei Pan. Near-linear algorithms for geometric hitting sets and set covers. *Discret. Comput. Geom.*, 63(2):460–482, 2020.

[Bak94] Brenda S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *J. ACM*, 41(1):153–180, 1994.

[BCL+16] Kevin Buchin, Daan Creemers, Andrea Lazzarotto, Bettina Speckmann, and Jules Wulms. Geo word clouds. In Chuck Hansen, Ivan Viola, and Xiaoru Yuan, editors, *Pacific Visualization Symposium (PacificVis'16)*, pages 144–151. IEEE, 2016.

[BDS16] Carla Binucci, Walter Didimo, and Enrico Spataro. Fully dynamic semantic word clouds. In Nikolaos G. Bourbakis, George A. Tsihrintzis, Maria Virvou, and Despina Kavraki, editors, *7th International Conference on Information, Intelligence, Systems & Applications, IISA 2016, Chalkidiki, Greece, July 13-15, 2016*, pages 1–6. IEEE, 2016.

[BETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[BF02] Hans L. Bodlaender and Fedor V. Fomin. Approximation of pathwidth of outerplanar graphs. *J. Algorithms*, 43(2):190–200, 2002.

[BFK+14] Lukas Barth, Sara Irina Fabrikant, Stephen G. Kobourov, Anna Lubiw, Martin Nöllenburg, Yoshio Okamoto, Sergey Pupyrev, Claudio Squarcella, Torsten Ueckerdt, and Alexander Wolff. Semantic word cloud representations: Hardness and approximation algorithms. In Alberto Pardo and Alfredo Viola, editors, *Theoretical Informatics (LATIN'14)*, volume 8392 of *Lecture Notes in Computer Science*, pages 514–525. Springer, 2014.

[BGL+21] Sujoy Bhore, Robert Ganian, Guangping Li, Martin Nöllenburg, and Jules Wulms. Worbel: Aggregating point labels into word clouds. In Xiaofeng Meng, Fusheng Wang, Chang-Tien Lu, Yan Huang, Shashi Shekhar, and Xing Xie, editors, *Advances in Geographic Information Systems (SIGSPATIAL'21)*, pages 256–267. ACM, 2021.

[BGPV08] Adam L. Buchsbaum, Emden R. Gansner, Cecilia Magdalena Procopiuc, and Suresh Venkatasubramanian. Rectangular layouts and contact graphs. *ACM Trans. Algorithms*, 4(1):8:1–8:28, 2008.

[BHS87] J Adrian Bondy, Glenn Hopkins, and William Staton. Lower bounds for induced forests in cubic graphs. *Can. Math. Bull.*, 30(2):193–199, 1987.

[Bie11] Therese C. Biedl. Small drawings of outerplanar graphs, series-parallel graphs, and other planar graphs. *Discret. Comput. Geom.*, 45(1):141–160, 2011.

[Bie15] Therese Biedl. On triangulating k-outerplanar graphs. *Discrete Applied Mathematics*, 181:275–279, 2015.

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

[BK16] Marthe Bonamy and Lukasz Kowalik. A 13k-kernel for planar feedback vertex set via region decomposition. *Theor. Comput. Sci.*, 645:25–40, 2016.

[BKP14] Lukas Barth, Stephen G. Kobourov, and Sergey Pupyrev. Experimental comparison of semantic word clouds. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms (SEA'14)*, volume 8504 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2014.

[BKZ15] Michael A. Bekos, Michael Kaufmann, and Christian Zielke. The book embedding problem from a sat-solving perspective. In Emilio Di Giacomo and Anna Lubiw, editors, *Graph Drawing and Network Visualization - 23rd International Symposium, GD 2015, Los Angeles, CA, USA, September 24-26, 2015, Revised Selected Papers*, volume 9411 of *Lecture Notes in Computer Science*, pages 125–138. Springer, 2015.

[BM88] Daniel Bienstock and Clyde L. Monma. On the complexity of covering vertices by faces in a planar graph. *SIAM Journal on Computing*, 17(1):53–76, 1988.

[Bod98] Hans L. Bodlaender. A partial $k$-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998.

[Bos] Mike Bostock. D3 data-driven documents. `https://d3js.org/`.

[BvDF+17] Michael A. Bekos, Thomas C. van Dijk, Martin Fink, Philipp Kindermann, Stephen G. Kobourov, Sergey Pupyrev, Joachim Spoerhase, and Alexander Wolff. Improved approximation algorithms for box contact representations. *Algorithmica*, 77(3):902–920, 2017.

[CCH+07] Amitabh Chaudhary, Danny Z. Chen, Xiaobo Sharon Hu, Michael T. Niemier, Ramprasad Ravichandran, and Kevin Whitton. Fabricatable interconnect and molecular QCA circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 26(11):1978–1991, 2007.

[CFK+15a] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[CFK+15b] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[CGMW09] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Christian Wolf. Inserting a vertex into a planar graph. In Claire Mathieu, editor, *Proc. 20th Symposium on Discrete Algorithms (SODA)*, pages 375–383. SIAM, 2009.

[CH16] Markus Chimani and Petr Hlinený. Inserting multiple edges into a planar graph. In Sándor P. Fekete and Anna Lubiw, editors, *Proc. 32nd International Symposium on Computational Geometry (SoCG)*, volume 51 of *LIPIcs*, pages 30:1–30:15, 2016.

[CPPS20] Fati Chen, Laurent Piccinini, Pascal Poncelet, and Arnaud Sallaberry. Node overlap removal algorithms: an extended comparative study. *J. Graph Algorithms Appl.*, 24(4):683–706, 2020.

[CWL+10] Weiwei Cui, Yingcai Wu, Shixia Liu, Furu Wei, Michelle X. Zhou, and Huamin Qu. Context-preserving, dynamic word cloud visualization. *IEEE Computer Graphics and Applications*, 30(6):42–53, 2010.

[DB] Jessica Davies and Fahiem Bacchus. A fast and robust maxsat solver.

[DEL11] Walter Didimo, Peter Eades, and Giuseppe Liotta. Drawing graphs with right angle crossings. *Theor. Comput. Sci.*, 412(39):5156–5166, 2011.

[DF01] Camil Demetrescu and Irene Finocchi. Removing cycles for minimizing crossings. *JEA*, 6:2, 2001.

[dFdMR94] Hubert de Fraysseix, Patrice Ossona de Mendez, and Pierre Rosenstiehl. On triangle contact graphs. *Combinatorics, Probability and Computing*, 3:233–246, 1994.

[DH05] Erik Demaine and Mohammad Hajiaghayi. Bidimensionality: New connections between FPT algorithms and PTASs. In *Proc. 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 590–601, 01 2005.

[Die17] Reinhard Diestel. *Graph Theory, Fifth Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2017.

[DLM19] Walter Didimo, Giuseppe Liotta, and Fabrizio Montecchiani. A survey on graph drawing beyond planarity. *ACM Comput. Surv.*, 52(1):4:1–4:37, 2019.

[DMW08] Tim Dwyer, Kim Marriott, and Michael Wybrow. Dunnart: A constraint-based network diagram authoring tool. In Ioannis G. Tollis and Maurizio Patrignani, editors, *Graph Drawing, 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008. Revised Papers*, volume 5417 of *Lecture Notes in Computer Science*, pages 420–431. Springer, 2008.

[dNE01] Hugo A. D. do Nascimento and Peter Eades. User hints for directed graph drawing. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23-26, 2001, Revised Papers*, volume 2265 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2001.

[EdMN95] Peter Eades and Candido F. X. de Mendonça N. Vertex splitting and tension-free layout. In Franz-Josef Brandenburg, editor, *Proc. 3rd International Symposium on Graph Drawing (GD)*, volume 1027 of *LNCS*, pages 202–211. Springer, 1995.

[EF10] Niklas Elmqvist and Jean-Daniel Fekete. Hierarchical Aggregation for Information Visualization: Overview, Techniques, and Design Guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, May 2010.

[EGH+20a] Eduard Eiben, Robert Ganian, Thekla Hamm, Fabian Klute, and Martin Nöllenburg. Extending nearly complete 1-planar drawings in polynomial time. In Javier Esparza and Daniel Král', editors, *Proc. 45th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 170 of *LIPIcs*, pages 31:1–31:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[EGH+20b] Eduard Eiben, Robert Ganian, Thekla Hamm, Fabian Klute, and Martin Nöllenburg. Extending partial 1-planar drawings. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *Proc. 47th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 168 of *LIPIcs*, pages 43:1–43:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[EKK+18] David Eppstein, Philipp Kindermann, Stephen G. Kobourov, Giuseppe Liotta, Anna Lubiw, Aude Maignan, Debajyoti Mondal, Hamideh Vosoughpour, Sue Whitesides, and Stephen K. Wismath. On the planar split thickness of graphs. *Algorithmica*, 80(3):977–994, 2018.

[EMW86] Peter Eades, Brendan McKay, and Nicholas Wormald. On an edge crossing problem. In *ACSC '86*, pages 327–334, 1986.

[ERG02] P. Eklund, N. Roberts, and S. Green. OntoRama: Browsing RDF ontologies using a hyperbolic-style browser. In *First International Symposium on Cyber Worlds, 2002. Proceedings.*, pages 405–411, November 2002.

[EW94] Peter Eades and Nicholas C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[FdFdMN98] Luérbio Faria, Celina M. H. de Figueiredo, and Candido F. X. de Mendonça N. The splitting number of the 4-cube. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *Proc. 3rd Latin American Symposium on Theoretical Informatics (LATIN)*, volume 1380 of *LNCS*, pages 141–150. Springer, 1998.

[FdFdMN01] Luérbio Faria, Celina M. H. de Figueiredo, and Candido F. X. de Mendonça N. Splitting number is NP-complete. *Discrete Applied Mathematics*, 108(1):65–83, 2001.

[Fei09] Jonathan Feinberg, 2009. `https://www.wordle.net/`.

[Fel13] Stefan Felsner. Rectangle and square representations of planar graphs. In János Pach, editor, *Thirty Essays on Geometric Graph Theory*, pages 213–248. Springer, 2013.

[FHSV16] Martin Fink, John Hershberger, Subhash Suri, and Kevin Verbeek. Bundled crossings in embedded graphs. In Evangelos Kranakis, Gonzalo Navarro, and Edgar Chávez, editors, *Proc. 12th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 9644 of *LNCS*, pages 454–468. Springer Berlin Heidelberg, 2016.

[Fra22] Fabrizio Frati. Planar rectilinear drawings of outerplanar graphs in linear time. *Computat. Geom.*, 103:101854, 2022.

[Fre96] Greg N. Frederickson. Searching among intervals and compact routing tables. *Algorithmica*, 15(5):448–466, 1996.

[GJ77] M. R. Garey and David S. Johnson. The rectilinear steiner tree problem is NP complete. *SIAM Journal of Applied Mathematics*, 32(4):826–834, 1977.

[GJ83] M. R. Garey and D. S. Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983.

[GKN04] Emden R. Gansner, Yehuda Koren, and Stephen C. North. Graph drawing by stress majorization. In János Pach, editor, *Graph Drawing (GD'04)*, volume 3383 of *Lecture Notes in Computer Science*, pages 239–250. Springer, 2004.

[GNV23] Martin Gronemann, Martin Nöllenburg, and Anaïs Villedieu. Splitting plane graphs to outerplanarity. In Chun-Cheng Lin, Bertrand M. T. Lin, and Giuseppe Liotta, editors, *WALCOM: Algorithms and Computation - 17th International Conference and Workshops, WALCOM 2023, Hsinchu, Taiwan, March 22-24, 2023, Proceedings*, volume 13973 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 2023.

[Har86] Nora Hartsfield. The toroidal splitting number of the complete graph $k_n$. *Discrete Mathematics*, 62(1):35–47, 1986.

[Har87] Nora Hartsfield. The splitting number of the complete graph in the projective plane. *Graphs and Combinatorics*, 3(1):349–356, 1987.

[HBF08] Nathalie Henry, Anastasia Bezerianos, and Jean-Daniel Fekete. Improving the readability of clustered social networks using node duplication. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1317–1324, 2008.

[Hic13] Leo Hickman. How algorithms rule the world. *The Guardian*, 2013.

[HJR85] Nora Hartsfield, Brad Jackson, and Gerhard Ringel. The splitting number of the complete graph. *Graphs and Combinatorics*, 1(1):311–329, 1985.

[HL05] Refael Hassin and Asaf Levin. A better-than-greedy approximation algorithm for the minimum set cover problem. *SIAM J. Comput.*, 35(1):189–200, 2005.

[HLSG07] Xiaodi Huang, Wei Lai, A. S. M. Sajeev, and Junbin Gao. A new algorithm for removing node overlapping in graph visualization. *Inf. Sci.*, 177(14):2821–2844, 2007.

[HPP+20] Marti A. Hearst, Emily Pedersen, Lekha Patil, Elsie Lee, Paul Laskowski, and Steven Franconeri. An evaluation of semantically grouped word cloud designs. *IEEE Trans. Vis. Comput. Graph.*, 26(9):2748–2761, 2020.

[HRD10] Nathalie Henry Riche and Tim Dwyer. Untangling euler diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1090–1099, 2010.

[HW+79] John A Hartigan, Manchek A Wong, et al. A k-means clustering algorithm. *Applied statistics*, 28(1):100–108, 1979.

[JdKW21] Bart M. P. Jansen, Jari J. H. de Kroon, and Michał Włodarczyk. Vertex deletion parameterized by elimination distance and even less. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, pages 1757–1769, New York, NY, USA, June 2021. Association for Computing Machinery.

[JLS13] Bart M. P. Jansen, Daniel Lokshtanov, and Saket Saurabh. A near-optimal planarization algorithm. In Chandra Chekuri, editor, *Proc. 2014 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Proceedings, pages 1802–1811. Society for Industrial and Applied Mathematics, 2013.

[JLS15] Jaemin Jo, Bongshin Lee, and Jinwook Seo. Wordleplus: Expanding wordle's use through natural interaction and animation. *IEEE Computer Graphics and Applications*, 35(6):20–28, 2015.

[JM96] Michael Jünger and Petra Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16(1):33–59, 1996.

[JM97] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *JGAA*, 1(1):1–25, 1997.

[JR84] Brad Jackson and Gerhard Ringel. The splitting number of complete bipartite graphs. *Archiv der Mathematik*, 42(2):178–184, 1984.

[Kan96] Goos Kant. Augmenting outerplanar graphs. *J. Algorithms*, 21(1):1–25, 1996.

[Kaw09] Ken-ichi Kawarabayashi. Planarity allowing few error vertices in linear time. In *Proc. 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 639–648, 2009.

[KBJM12] Martin Krzywinski, Inanc Birol, Steven J.M. Jones, and Marco A. Marra. Hive plots—rational approach to visualizing networks. *Briefings in Bioinformatics*, 13(5):627–644, 2012.

[KDMW16] Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. HOLA: human-like orthogonal network layout. *IEEE Trans. Vis. Comput. Graph.*, 22(1):349–358, 2016.

[KFS+22] Minoru Kanehisa, Miho Furumichi, Yoko Sato, Masayuki Kawashima, and Mari Ishiguro-Watanabe. KEGG for taxonomy-based analysis of pathways and genomes. *Nucleic Acids Research*, page gkac963, October 2022.

[KK85] Krzysztof Kozminski and Edwin Kinnen. Rectangular duals of planar graphs. *Networks*, 15(2):145–157, 1985.

[KL17] Tom Kelly and Chun-Hung Liu. Minimum size of feedback vertex sets of planar graphs of girth at least five. *Eur. J. Comb.*, 61:138–150, 2017.

[KLKS10] Kyle Koh, Bongshin Lee, Bo Hyoung Kim, and Jinwook Seo. Maniwordle: Providing flexible control over wordle. *IEEE Trans. Vis. Comput. Graph.*, 16(6):1190–1197, 2010.

[KLL02] Ton Kloks, C.M. Lee, and Jiping Liu. New algorithms for k-face cover, k-feedback vertex set, and k-disjoint cycles on plane and planar graphs. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Luděk Kučera, editors, *Proc. 28th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, LNCS, pages 282–295. Springer, 2002.

[KLP+15] Eun Jung Kim, Alexander Langer, Christophe Paul, Felix Reidl, Peter Rossmanith, Ignasi Sau, and Somnath Sikdar. Linear kernels and single-exponential algorithms via protrusion decompositions. *ACM Transactions on Algorithms*, 12(2):21:1–21:41, 2015.

[Koe36] Paul Koebe. Kontaktprobleme der konformen Abbildung. *Ber. Sächs. Akad. Wiss. Leipzig, Math.-Phys. Klasse*, 88:141–164, 1936.

[KT15] Yasuaki Kobayashi and Hisao Tamaki. A fast and simple subexponential fixed parameter algorithm for one-sided crossing minimization. *Algorithmica*, 72(3):778–790, 2015.

[KU16] Kolja B. Knauer and Torsten Ueckerdt. Three ways to cover a graph. *Discrete Mathematics*, 339(2):745–758, 2016.

[LDY18] Chenlu Li, Xiaoju Dong, and Xiaoru Yuan. Metro-wordle: An interactive visualization for urban text distributions based on wordle. *Vis. Informatics*, 2(1):50–59, 2018.

[LHT17] Antoine Lhuillier, Christophe Hurter, and Alexandru C. Telea. State of the art in edge and trail bundling techniques. *Computer Graphics Forum*, 36(3):619–645, 2017.

[Lie01] Annegret Liebers. Planarizing graphs - A survey and annotated bibliography. *Journal of Graph Algorithms and Applications*, 5(1):1–74, 2001.

[LL96] William Lenhart and Giuseppe Liotta. Proximity drawings of outerplanar graphs. In Stephen C. North, editor, *Proc. 4th International Symposium on Graph Drawing (GD)*, volume 1190 of *LNCS*, pages 286–302. Springer, 1996.

[LLL19] Sylvain Lazard, William J. Lenhart, and Giuseppe Liotta. On the edge-length ratio of outerplanar graphs. *Theor. Comput. Sci.*, 770:88–94, 2019.

[LPP+06] Bongshin Lee, C.S. Parr, C. Plaisant, B.B. Bederson, V.D. Veksler, W.D. Gray, and C. Kotfila. TreePlus: Interactive Exploration of Networks with Enhanced Tree Layouts. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1414–1426, November 2006. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[LSDK18] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph Summarization Methods and Applications: A Survey. *ACM Computing Surveys*, 51(3):62:1–62:34, June 2018.

[LY80] John M. Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *J. Comput. Syst. Sci.*, 20(2):219–230, 1980.

[LZ96] Jiping Liu and Cheng Zhao. A new bound on the feedback vertex sets in cubic graphs. *Discret. Math.*, 148(1-3):119–131, 1996.

[MELS95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[MFFP11] Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Generalized louvain method for community detection in large networks. In Sebastián Ventura, Ajith Abraham, Krzysztof J. Cios, Cristóbal Romero, Francesco Marcelloni, José Manuel Benítez, and Eva Lucrecia Gibaja Galindo, editors, *11th International Conference on Intelligent Systems Design and Applications, ISDA 2011, Córdoba, Spain, November 22-24, 2011*, pages 88–93. IEEE, 2011.

[Moh] Alva Mohanda. Harry potter and the philosopher's stone summary. `https://bookanalysis.com/jk-rowling/harry-potter-and-the-philosophers-stone/summary/`.

[Moh01] Bojan Mohar. Face covers and the genus problem for apex graphs. *J. Comb. Theory, Ser. B*, 82(1):102–117, 2001.

[MP15] Dániel Marx and Michal Pilipczuk. Optimal parameterized algorithms for planar facility location problems using voronoi diagrams. *CoRR*, abs/1504.05476, 2015.

[MRA+21] Fintan McGee, Benjamin Renoust, Daniel Archambault, Mohammad Ghoniem, Andreas Kerren, Bruno Pinaud, Margit Pohl, Benoit Otjacques, Guy Melançon, and Tatiana von Landesberger. *Visual Analysis of Multilayer Networks*. Morgan & Claypool, 2021.

[MS12] Dániel Marx and Ildikó Schlotter. Obtaining a planar graph by vertex deletion. *Algorithmica*, 62(3-4):807–822, 2012.

[MZ99] Anil Maheshwari and Norbert Zeh. External memory algorithms for outerplanar graphs. In Alok Aggarwal and C. Pandu Rangan, editors, *Proc. 10th International Symposium on Algorithms and Computation (ISAAC)*, volume 1741 of *LNCS*, pages 307–316. Springer, 1999.

[NDG+17] Alberto Noronha, Anna Dröfn Daníelsdóttir, Piotr Gawron, Freyr Jóhannsson, Soffía Jónsdóttir, Sindri Jarlsson, Jón Pétur Gunnarsson, Sigurður Brynjólfsson, Reinhard Schneider, Ines Thiele, and Ronan M T Fleming. ReconMap: an interactive visualization of human metabolism. *Bioinformatics*, 33(4):605–607, February 2017.

[NK16] Sabrina Nusrat and Stephen Kobourov. The state of the art in cartograms. *Computer Graphics Forum*, 35(3):619–642, 2016.

[NOM+19] Sune S. Nielsen, Marek Ostaszewski, Fintan McGee, David Hoksza, and Simone Zorzan. Machine learning to support the presentation of complex pathway graphs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 18(3):1130–1141, 2019.

[NSS01] Assaf Natanzon, Ron Shamir, and Roded Sharan. Complexity classification of some edge modification problems. *Discret. Appl. Math.*, 113(1):109–128, 2001.

[NST+22] Martin Nöllenburg, Manuel Sorge, Soeren Terziadis, Anaïs Villedieu, Hsiang-Yun Wu, and Jules Wulms. Planarizing graphs and their drawings by vertex splitting. In Patrizio Angelini and Reinhard von Hanxleden, editors, *Proc. 30th International Symposium on Graph Drawing and Network Visualization (GD'22)*, LNCS. Springer, 2022.

[NVW21] Martin Nöllenburg, Anaïs Villedieu, and Jules Wulms. Layered area-proportional rectangle contact representations. In Helen C. Purchase and Ignaz Rutter, editors, *Proc. 29th International Symposium Graph Drawing and Network Visualization (GD)*, volume 12868 of *LNCS*, pages 318–326. Springer, 2021.

[Nö20] Martin Nöllenburg. Crossing layout in non-planar graphs. In Seok-Hee Hong and Takeshi Tokuyama, editors, *Beyond Planar Graphs*, chapter 11, pages 187–209. Springer Nature Singapore, 2020.

[Oka20] Yoshio Okamoto. Angular resolutions: Around vertices and crossings. In Seok-Hee Hong and Takeshi Tokuyama, editors, *Beyond Planar Graphs*, chapter 10, pages 171–186. Springer Nature Singapore, 2020.

[PBHI+] Hrishikesh Paul, Katy Börner, Bruce W Herr II, Ellen M Quardokus, Sai Ajay Vutukuri, Mac Vogelsang, and Nikhil Mahadevaswamy. Hubmap ccf asct+b reporter.

[PCA02] Helen C. Purchase, David A. Carrington, and Jo-Anne Allder. Empirical evaluation of aesthetics-based graph layout. *Empir. Softw. Eng.*, 7(3):233–255, 2002.

[PFH+18] Nicola Pezzotti, Jean-Daniel Fekete, Thomas Höllt, Boudewijn P. F. Lelieveldt, Elmar Eisemann, and Anna Vilanova. Multiscale visualization and exploration of large bipartite graphs. *CGF*, 37(3):549–560, 2018.

[PPP12] Helen C. Purchase, Christopher Pilcher, and Beryl Plimmer. Graph drawing aesthetics—created by users, not algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 18(1):81–92, 2012.

[PRS98] Doowon Paik, Sudhakar M. Reddy, and Sartaj Sahni. Vertex splitting in dags and applications to partial scan designs and lossy circuits. *Int. J. Found. Comput. Sci.*, 9(4):377–398, 1998.

[PT97] János Pach and Géza Tóth. Graphs drawn with few crossings per edge. *Comb.*, 17(3):427–439, 1997.

[Pur00] Helen C. Purchase. Effective information visualisation: a study of graph drawing aesthetics and algorithms. *Interact. Comput.*, 13(2):147–162, 2000.

[RS95] N. Robertson and P. D. Seymour. Graph minors. xiii. the disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[Sch90] Walter Schnyder. Embedding planar graphs on the grid. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 138–148. SIAM, 1990.

[Sch08] John Schwenkler. Portrait of the candidate as a pile of words, 2008. `http://archive.boston.com/bostonglobe/ideas/articles/2008/08/03/portrait_of_the_candidate_as_a_pile_of_words/`.

[Sch18] Marcus Schaefer. *Crossing Numbers of Graphs*. CRC Press, 2018.

[Sch21] Marcus Schaefer. Rac-drawability is $\exists \diagdown$-complete. In Helen C. Purchase and Ignaz Rutter, editors, *Graph Drawing and Network Visualization - 29th International Symposium, GD 2021, Tübingen, Germany, September 14-17, 2021, Revised Selected Papers*, volume 12868 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2021.

[Sip97] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[SKK+08] Christin Seifert, Barbara Kump, Wolfgang Kienreich, Gisela Granitzer, and Michael Granitzer. On the beauty and usability of tag clouds. In *Information Visualisation (IV'08)*, pages 17–25. IEEE Computer Society, 2008.

[SS10] Pascal Schweitzer and Patrick Schweitzer. Connecting face hitting sets in planar graphs. *Inf. Process. Lett.*, 111(1):11–15, 2010.

[SST04] Ron Shamir, Roded Sharan, and Dekel Tsur. Cluster graph modification problems. *Discret. Appl. Math.*, 144(1-2):173–182, 2004.

[SSW+17] Erich Schubert, Andreas Spitz, Michael Weiler, Johanna Geiß, and Michael Gertz. Semantic word clouds with background corpus normalization and t-distributed stochastic neighbor embedding. *CoRR*, abs/1708.03569, 2017.

[ST94] Paul D. Seymour and Robin Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *TSMC*, 11(2):109–125, 1981.

[SZH11] Celine Scornavacca, Franziska Zickmann, and Daniel H. Huson. Tanglegrams for rooted phylogenetic trees and networks. *Bioinformatics*, 27(13):i248–i256, 2011.

[TH79] William T. Trotter and Frank Harary. On double and multiple interval graphs. *J. Graph Theory*, 3(3):205–211, 1979.

[UEM11] Chris Umbel, Rob Ellis, and Russell Mull. Natural, 2011. `https://www.npmjs.com/package/natural`.

[UKG88] Shuichi Ueno, Yoji Kajitani, and Shin'ya Gotoh. On the nonseparating independent set problem and feedback set problem for graphs with no vertex degree exceeding three. *Discret. Math.*, 72(1-3):355–360, 1988.

[Vin23] James Vincent. AI art tools Stable Diffusion and Midjourney targeted with copyright lawsuit, January 2023.

[vKS07] Marc van Kreveld and Bettina Speckmann. On rectangular cartograms. *Computational Geometry*, 37(3):175–187, 2007.

[vLKS+11] Tatiana von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J. van Wijk, Jean-Daniel Fekete, and Dieter W. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Computer Graphics Forum*, 30(6):1719–1749, 2011.

[VW08] Fernanda B. Viégas and Martin Wattenberg. Timelines - tag clouds and the case for vernacular visualization. *Interactions*, 15(4):49–52, 2008.

[VWF09] Fernanda B. Viégas, Martin Wattenberg, and Jonathan Feinberg. Participatory visualization with wordle. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1137–1144, 2009.

[WCB+18] Yunhai Wang, Xiaowei Chu, Chen Bao, Lifeng Zhu, Oliver Deussen, Baoquan Chen, and Michael Sedlmair. Edwordle: Consistency-preserving word cloud editing. *IEEE Trans. Vis. Comput. Graph.*, 24(1):647–656, 2018.

[WCZ+20] Yunhai Wang, Xiaowei Chu, Kaiyi Zhang, Chen Bao, Xiaotong Li, Jian Zhang, Chi-Wing Fu, Christophe Hurter, Bongshin Lee, and Oliver Deussen. Shapewordle: Tailoring wordles using shape-aware archimedean spirals. *IEEE Trans. Vis. Comput. Graph.*, 26(1):991–1000, 2020.

[wik22] European union, 2022. `https://en.wikipedia.org/wiki/European_Union`.

[WNSV19] Hsiang-Yun Wu, Martin Nöllenburg, Filipa L. Sousa, and Ivan Viola. Metabopolis: Scalable network layout for biological pathway diagrams in urban map style. *BMC Bioinformatics*, 20(1):1–20, 2019.

[WNV20] Hsiang-Yun Wu, Martin Nöllenburg, and Ivan Viola. Multi-level area balancing of clustered graphs. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–15, 2020.

[WPW+11] Yingcai Wu, Thomas Provan, Furu Wei, Shixia Liu, and Kwan-Liu Ma. Semantic-preserving word clouds by seam carving. *Comput. Graph. Forum*, 30(3):741–750, 2011.

[XTL16] Jin Xu, Yubo Tao, and Hai Lin. Semantic word cloud generation based on word embeddings. In Chuck Hansen, Ivan Viola, and Xiaoru Yuan, editors, *Pacific Visualization (PacificVis'16)*, pages 239–243. IEEE Computer Society, 2016.

[Yan78] Mihalis Yannakakis. Node-and edge-deletion NP-complete problems. In *Proc. 10th Annual ACM Symposium on Theory of Computing (STOC)*, pages 253–264. ACM, 1978.

[YS93] Kok-Hoo Yeap and Majid Sarrafzadeh. Floor-planning by graph dualization: 2-concave rectilinear modules. *SIAM J. Comput.*, 22(3):500–526, 1993.

[ZXYQ13] Hong Zhou, Panpan Xu, Xiaoru Yuan, and Huamin Qu. Edge bundling in information visualization. *Tsinghua Science and Technology*, 18(2):145–156, 2013.

# Publications List

- **MySemCloud: Semantic-aware Word Cloud Editing.** Michael Huber, Martin Nöllenburg, Anaïs Villedieu. In *Proceedings of the 16th IEEE Pacific Visualization Symposium (PacificVis 2023)*, Seoul, South Korea, April 2023.

- **Splitting Plane Graphs to Outerplanarity**. Martin Gronemann, Martin Nöllenburg, Anaïs Villedieu. In *Proceedings of the 17th International Conference and Workshops on Algorithms and Computation (WALCOM 2023)*, Hsinchu, Taiwan, March 2023.

- **Splitting Vertices in 2-Layer Graph Drawings.** Reyan Ahmed, Patrizio Angelini, Michael A. Bekos, Giuseppe Di Battista, Michael Kaufmann, Philipp Kindermann, Stephen G. Kobourov, Martin Nöllenburg, Antonios Symvonis, Anaïs Villedieu, Markus Wallinger. In *IEEE Computer Graphics and Applications (CGA)*, 2023.

- **Minimum Link Fencing.** Sujoy Bhore, Fabian Klute, Maarten Löffler, Martin Nöllenburg, Soeren Terziadis, Anaïs Villedieu. In *Proceedings of the 33rd International Symposium on Algorithms and Computation (ISAAC 2022)*, Seoul, South Korea, December 2022.

- **Turbocharging Heuristics for Weak Coloring Numbers.** Alexander Dobler, Manuel Sorge, Anaïs Villedieu. Turbocharging Heuristics for Weak Coloring Numbers. In *Proceedings of the 30th Annual European Symposium on Algorithms (ESA 2022)*, Potsdam, Germany, September 2022.

- **Planarizing Graphs and Their Drawings by Vertex Splitting.** Martin Nöllenburg, Manuel Sorge, Soeren Terziadis, Anaïs Villedieu, Hsiang-Yun Wu, Jules Wulms. In *Proceedings of the 30th International Symposium on Graph Drawing and Network Visualization (GD 2022)*, Tokyo, Japan, September 2022.

- **Multidimensional Manhattan Preferences.** Jiehua Chen, Martin Nöllenburg, Sofia Simola, Anaïs Villedieu, Markus Wallinger. In *Proceedings of the 15th Latin American Symposium (LATIN 2022)*, Guanajuato, Mexico, November 2022.

- **Layered Area-Proportional Rectangle Contact Representations.** Martin Nöllenburg, Anaïs Villedieu, Jules Wulms. In *Proceedings of the 29th International Symposium on Graph Drawing and Network Visualization (GD 2021)*, Tübingen, Germany, September 2021.

- **Efficient non-segregated routing for reconfigurable demand-aware networks.** Thomas Fenz, Klaus-Tycho Foerster, Stefan Schmid, Anaïs Villedieu. In *Computer Communications (COMCOM)*, 2020.