FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Engineering an Algorithm for Strict Outerconfluent Graph Drawings

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Buchta, BSc.
Matrikelnummer 01425583

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg
Mitwirkung: Fabian Klute, MSc. BSc.

Wien, 9. Oktober 2019

Markus Buchta

Martin Nöllenburg

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

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Engineering an Algorithm for Strict Outerconfluent Graph Drawings

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Markus Buchta, BSc.
Registration Number 01425583

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg
Assistance: Fabian Klute, MSc. BSc.

Vienna, 9th October, 2019

Markus Buchta

Martin Nöllenburg

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

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Markus Buchta, BSc.

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, 9. Oktober 2019

Markus Buchta

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich danke meinen Betreuern Martin Nöllenburg und Fabian Klute für die wunderbare Betreuung meiner Diplomarbeit. Ich hatte durchwegs das Gefühl, dass beide Interesse sowohl am Thema als auch an dem Erfolg der Arbeit hatten. Das spiegelte sich vorallem in der ausgezeichneten Kommunikation wider. Bei Problemen oder Fragen konnte ich beide jederzeit kontaktieren und erhielt die Antwort meist innerhalb einer Stunde. Ich bewundere das fachliche Know-How beider und kann in ehrfurcht sagen, dass ich mir keine bessere Betreeung hätte wünschen können.

Des Weiteren möchte ich meiner Freundin Viola Ehrnhofer für ihre Unterstützung während des gesamten Studiums danken. Selbst an schlechten Tagen hat sie es geschafft mich aufzuheitern. Ich möchte ihr auch dafür danken, dass sie mich überhaupt dazu ermutigt hat zu studieren. Des Weiteren hat sie mir während der Prüfungszeiten meinen nötigen Freiraum gegeben und Rücksicht auf mich genommen, wodurch ich gute Leistungen erzielen konnte.

Ebenfalls möchte ich meiner Mutter Christa Buchta danken, welche es mir ermöglichte zu studieren. Sie hat mich während meiner gesamten Ausbildung, von der Volksschule bis zum Studium, stets unterstützt und es mir ermöglicht meine Wünsche und Träume zu realisieren.

Ein weiteres Dank geht an alle StudienkollegenInnen, durch welche das Studium immer sehr unterhaltsam und spannend war. Das gilt sowohl für die KollegInnen die ich vor dem Studium schon kannte als auch jene tollen Bekanntschaften die ich währenddessen schloss. Ich habe das gemeinsame Studieren mit ihnen genossen und werde die Zeit sicher vermissen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I thank my supervisors Martin Nöllenburg and Fabian Klute for the wonderful supervision of my diploma thesis. I had the feeling that both were interested in the topic as well as in the success of the thesis. This was reflected above all in the excellent communication. In case of problems or questions, I could contact both of them at any time and usually received the answer within an hour. I admire their technical know-how and can say in awe that I could not have wished for better support.

Furthermore, I would like to thank my friend Viola Ehrnhofer for her support throughout my study. Even on bad days she managed to cheer me up. I would also like to thank her for encouraging me to study at all. She also gave me the freedom and consideration I needed during my exams, which allowed me to perform well.

I would also like to thank my mother Christa Buchta, who made it possible for me to study. She has supported me throughout my education, from elementary school to university, and has made it possible for me to realize my wishes and dreams.

Further thanks go to all my fellow students who made my studies so entertaining and exciting. This applies to the colleagues I already knew before my studies as well as to the great acquaintances I made during my studies. I enjoyed studying together with them and will certainly miss the time.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Eine konfluente Zeichnung eines Graphens ist eine Zeichnung, in welcher die Knoten des Graphens als Punkte in der Ebene und die Kanten als glatte Pfade durch ein planares System von Kreuzungen und Bögen dargestellt werden. Als außen-konfluente Zeichnung werden jene konfluente Zeichnungen bezeichnet bei welchen alle Knoten auf der Außenfläche liegen. Schlussendlich ist eine strikte außen-konfluente Zeichnung eine außen-konfluente Zeichnung bei der es keine zwei unterschiedliche Pfade zwischen zwei Knoten gibt und kein Knoten einen Pfad zu sich selbst aufweist. Eppstein et al.(2016) beschrieben auf einem sehr abstrakten Level einen Algorithmus, welcher für einen gegebenen Graphen zusammen mit einer Ordnung der Knoten feststellt ob dieser Graph mit dieser Ordnung eine strikt außen-konfluente Zeichnung besitzt oder nicht. Bisher wurde dieser Algorithmus jedoch noch nicht implementiert. Ein Teil meiner Arbeit ist die Implementierung dieses Algorithmus sowie eine detailierte Beschreibung des Algorithmus. Mit der korrekten Implementierung des Algorithmus war es uns dann möglich zu testen ob unterschiedliche Graphklassen solche Zeichnungen besitzen oder nicht. Ein Ergebnis dieser Auswertung ist, dass bipartite Permutationsgraphen nicht immer eine solche Zeichnung besitzen, obwohl dies unsere Annahme war.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

A confluent drawing is a drawing of a graph where vertices are presented as points in the plane and edges as smooth paths through a planar system of junctions and arcs. An outerconfluent drawing is a confluent drawing in which all vertices of the graph lie on the boundary of the outer face of the drawing. Furthermore, a strict outerconfluent drawing is an outerconfluent drawing with the additional constraint that there is no self-loop between two vertices and that there exists at most one path between two vertices. Eppstein et al.(2016) abstractly defined an algorithm for checking a given graph together with a vertex order for a strict outerconfluent drawing. However, this algorithm has never been implemented. As part of this thesis the algorithm was implemented and a detailed description of each individual step is presented. With the correct implementation we tried to find different classes of graphs which do or do not obtain strict outerconfluent drawings. One of our results is that bipartite permutation graphs do not obtain strict outerconfluent drawings although this was suspected in a recent paper by Förster et al.(2019).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Contribution 3

2 Preliminaries **5**
2.1 Graphs 5
2.2 Strict outerconfluent graph drawings 6
2.3 Graph classes 8

3 Description of the algorithm **11**
3.1 Lookup table T 13
3.2 Lookup tables N 16
3.3 Discovery of the funnels 17
3.4 Creation of the junction skeleton 20
3.5 Creation of the canonical diagram 28

4 Implementation **35**
4.1 Implementation process 35
4.2 Implementation details 38
4.3 Problems during implementation 40

5 Counterexample for bipartite permutation graphs **43**

6 Conclusion **47**

List of Figures **49**

List of Algorithms **51**

Bibliography **53**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

A graph consists of vertices, often representing certain objects, and edges connecting vertices, representing relations between vertices. There exist different ways of representing graphs such as adjacency matrices, sets of vertices and edges or vertices as points and edges as curves in the plane (i.e. a graph drawing or node-link diagram). Encoding information in a graph drawing as well as reading information from graph drawings is part of our daily life. Every city plan, metro plan, social media network or even family trees are depicted as graph drawings. Since we use graph drawings more often than we might perceive, it is important that graph drawings are easy to understand while still providing the necessary information. Frequently the understandability of graph drawings was defined by different criteria. One of the most important of these criteria is that the number of edge crossings should be as small as possible [PCJ97].

Minimizing the number of crossings is a NP-hard problem in many settings [Sch13]. Furthermore, for drawings in the plane, non-planar graphs cannot always be drawn without crossings. Since dense graphs have many crossings the idea of edge bundling techniques was born [HvW09]. The key idea of edge bundling techniques is to reduce edge clutter by finding subsets of edges with similar trajectories and grouping them into bundles. Examples of these edge bundling techniques are power graphs, metro-style bundling, or confluent drawings [ZPYQ13][BRH+17]. So called confluent drawings, introduced by Dickerson et al. in [DEGM05], allow to draw certain non-planar graphs in a planar way. This is achieved by merging groups of edges into so called tracks (like train tracks [HSS04]), such that crossing edges will turn into overlapping paths. The advantage of confluent drawings against other edge bundeling techniques is, that confluent drawings do not imply false neighours. A confluent drawing typically consists of vertices, arcs and junctions. The Kuratowski minors K_5 and $K_{3,3}$ then can be drawn in a planar way, as seen in Figure 1.1(a) and 1.1(b). It is important to mention that not every non-planar graph has a confluent drawing. For computing confluent drawings Dickerson et al. [DEGM05] described two heuristics (one for directed and one for undirected

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

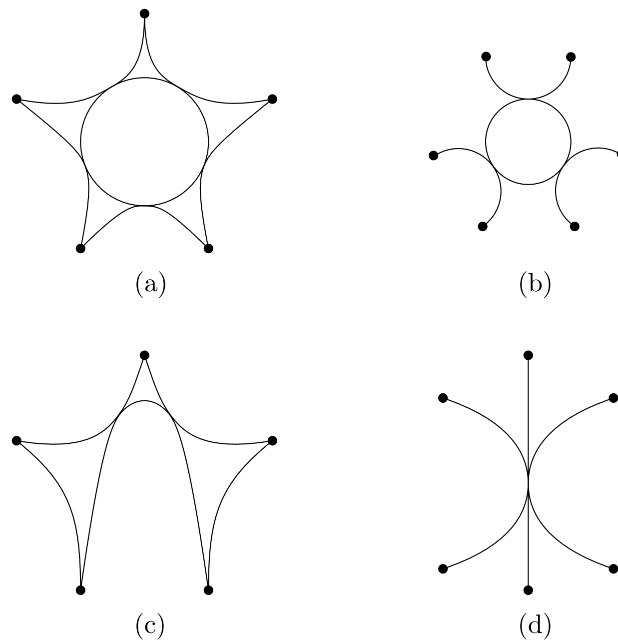


Figure 1.1: (a) A confluent drawing of K_5 (b) A confluent drawing of $K_{3,3}$ (c) A strict outerconfluent drawing of K_5 (d) A strict outerconfluent drawing of $K_{3,3}$

graphs) to check whether a graph has a confluent drawing or not. The main idea of the heuristics is to replace cliques and bicliques by so called traffic circles, which can be seen in Figure 1.1(a) and 1.1(b). They also showed that every graph of certain graph classes (i.e. cographs, complements of trees) admit a confluent drawing. Bach et al. [BRH+17] studied which of these edge bundling techniques is the easiest understandable for network visualization, which tend to be huge graphs. The result was that the confluent drawing style is easier to understand than the other techniques and therefore confluent drawings are more desirable.

Furthermore, Hui et al. [HSS04] introduced tree-confluent graphs. In these closed curves are not allowed. The traffic circle of $K_{3,3}$ in Figure 1.1(b) for example has a closed curve. Tree-confluent graphs therefore are a proper subset of confluent graphs. The concept of tree-confluent drawing was later generalized by Eppstein et al. [EGM05] to so called delta-confluent drawings. This was done by introducing a new type of junctions, so called delta-junctions. In a delta junction three paths are incoming and each path has a track to both other paths. Delta-confluent drawings consist of delta-junctions and so called lambda-junctions, in which two paths are merging into one. The main result of their work is, that distance-hereditary graphs coincide with the graphs that have delta-confluent drawings.

In 2015 outerplanar strict confluent drawings were introduced by Eppstein et al. [EHL+16]. Here we call them strict outerconfluent drawings to conform with latest literature [FGKN19]. An outerconfluent drawing is a confluent drawing in which the vertices are

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

placed on the boundary of a topological disk and all edges are drawn inside the disk. Strict outerconfluent drawings are outerconfluent drawings but with only one path between two adjacent vertices and each vertex is not adjacent to itself, which might make the drawings easier understandable than general (outer)confluent drawings. In their work Eppstein et al. [EHL+16] presented an algorithm for detecting whether a given graph together with a vertex order has a strict outerconfluent drawing or not. If there exists such a drawing the algorithm will return a data structure representing the drawing. Up to this point the algorithm has only been described as a theoretical result and not been implemented in any form. A correct implementation could not only be used to generate drawings, but also serve as a useful tool to investigate the still open questions. One of these questions is, if finding a strict outerconfluent drawing for a given graph is poly-time. This might be the case if certain patterns in the vertex order lead to such an drawing. Another open question is, which graph classes have or do not have such a drawing.

1.1 Contribution

In this thesis the algorithm sketched by Eppstein et al. is implemented. Since the description of the algorithm in [EHL+16] is given on a high-level, most of the steps must be refined. Therefore, each step of the refined algorithm will be described in more detail and with additional pseudo code. This description will be independent of any technologies and can be found in Chapter 3.

The implementation process used a test-driven approach, which can be applied since each step of the algorithm is deterministic. The idea behind the implementation process and why we consider that it led to a correct implementation of the algorithm, will also be explained in this thesis. Considering that the implementation was written in Java, there are some programming language specific features which were used for different steps. Further information about the implementation process and implementation details can be found in Chapter 4.

The last part of this thesis focuses on the evaluation of graph classes in context of finding graph classes admitting a strict outerconfluent drawing or not. A graph class is a set of graphs which fulfil certain properties. For example, the class of bipartite permutation graphs have outerconfluent drawings, but it is unknown if they also have strict outerconfluent drawings [FGKN19]. We present a counterexample for this and other classes. The results of this evaluation can be found in Chapter 5.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Preliminaries

This chapter covers the basics of graphs, graph drawings and strict outerconfluent drawings. Furthermore, the definitions of different important graph classes are given.

2.1 Graphs

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E, where each element in E is a tuple (u, v) of vertices $u, v \in V$. We say two vertices $u, v \in V$ are *adjacent* if there exists an edge $(u, v) \in E$. A graph is *undirected* if the existence of an edge (u, v) implies the existence of an edge (v, u), and *directed* otherwise. We call a graph *simple* if there are no edges $(u, u) \in E$, i.e. no self-loops are allowed. Unless otherwise stated all graphs considered throughout this work are simple and undirected. Let $G = (V, E)$ be a graph, then a *drawing* of G in the Euclidean plane is a mapping from vertices in V to points in the plane and from edges in E to simple curves in the plane, such that a curve representing an edge has only its start- and end-point identified with its vertices and no two curves cross in more than one interior point. A drawing of a graph G is called *plane* if no two curves cross aside from their end-points. A *face* in a plane drawing is an area surrounded by edges of G. The set of all faces is denoted by F. We can identify a plane drawing by using Eulers formula, which defines the relation between the number of edges, vertices and faces in the drawing as $|V| - |E| + |F| = 2$. A graph having a plane drawing is called *planar*.

For an *outerplanar drawing* it holds that the vertices can be placed on the boundary of a topological disk and all edges only pass through the inside of the disk. In other words in an outerplanar drawing there is no vertex placed inside a face. For an outerplanar drawing we have a ordering of vertices called π, representing the order of vertices along the boundary of the disk in clockwise direction. Furthermore $[a, b]$ is defined as the set of all vertices from a to b in π, inclusive a and b.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2 Strict outerconfluent graph drawings

Canonical diagram A *canonical diagram* $D = (N, J, \Gamma, F)$ was defined by Eppstein et al. in [EHL+16] as a set of *outer nodes* N, a set of points called *junctions*, a set of smooth curves calles *arcs* Γ and a set of *marked faces* F. In D a so called *trail* is a smooth curve from one outer node to another outer node following the arcs. Furthermore a trail can pass along sharp corners of marked faces. Following additional constraints also have to hold according to Eppstein et al. ([EHL+16] page 33):

1. Every arc is part of at least one trail.
2. Any two trails between the same two vertices must follow the same sequence of arcs and faces.
3. Each marked face must have at least four angles, and all its angles must be sharp.
4. Each arc must have either sharp angles or vertices at both of its ends.
5. For each junction j with exactly two arcs in each direction, let f and f' be the two faces with sharp angle are j. Then it is not allowed for f and f' to both be either marked or a triangle.

The number of internal faces, junctions and arcs of a canonical diagram have an upper bound according to Lemma 1.

Lemma 1 (Lemma 5 in [EHL+16]) *Every outerplanar strict confluent drawing has at most $n - 2$ internal face, $3n - 6$ junctions, and $5n - 9$ arc.*

Following Lemma specifies the relation between canonical diagrams and strict outerconfluent drawings.

Theorem 1 (Theorem 3 in [EHL+16]) *A graph G may be represented by a canonical diagram if and only if it may be represented by an outerplanar strict confluent drawing.*

Funnel Let j be a junction of the canonical diagram, then a *funnel* is a 4-tuple of vertices (a, b, c, d) where a is the vertex reached by a path that leaves j in one direction and goes as far clockwise as possible, b is the most counterclockwise vertex reachable in the same direction as a, c is the most clockwise vertex reachable in the other direction, and d is the most counterclockwise vertex reachable in the same direction as d. The circular intervals of vertices $[a, b]$ and $[c, d]$ are called *funnel intervals*. A circular interval $[a, b]$ is *separated* if either (1) a and b are not adjacent in G or (2) there exists a funnel f' with funnel intervals $[a, e]$ and $[f, b]$ where $e, d \in [a, b]$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

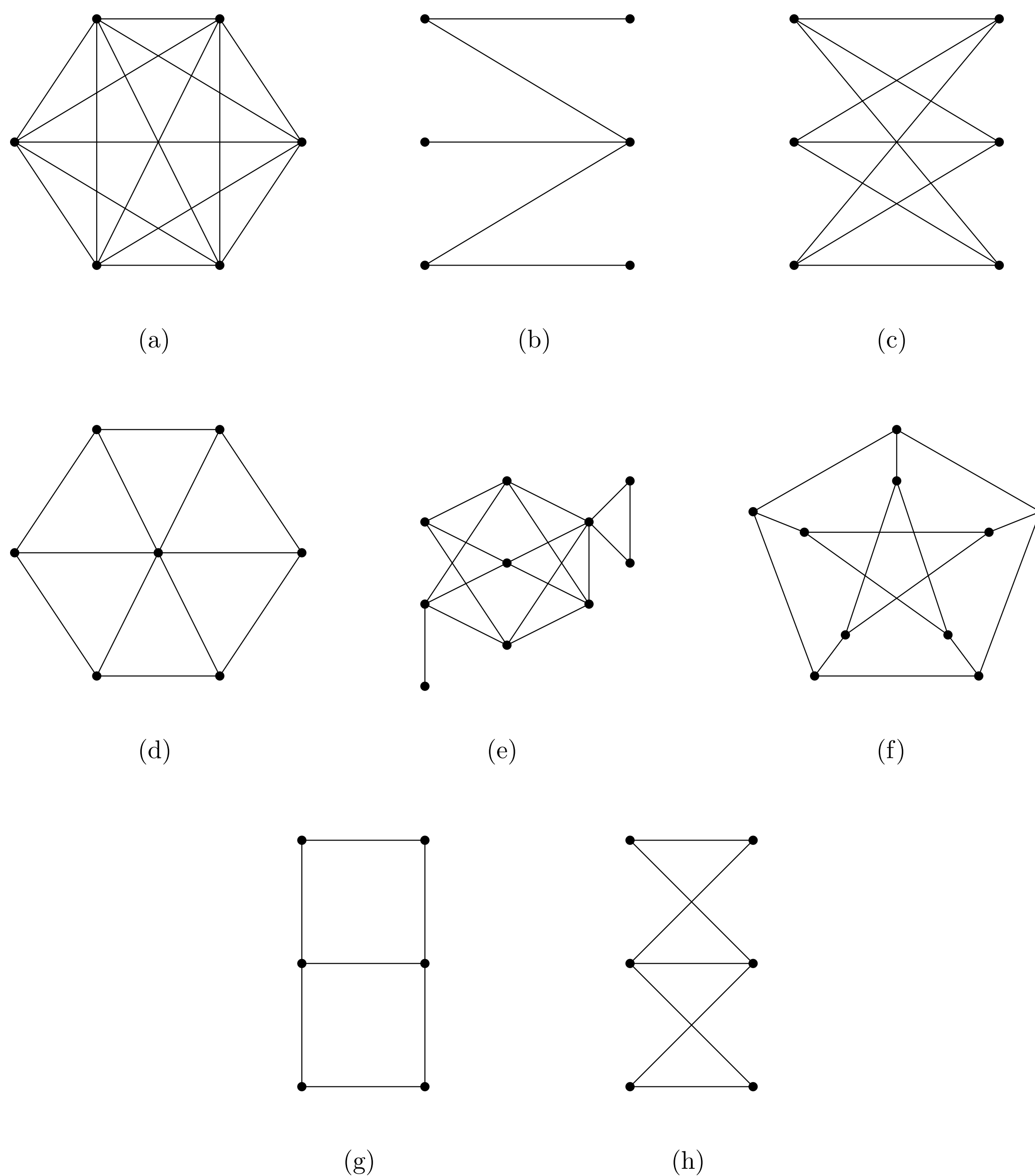


Figure 2.1: (a) K_6 (b) An arbitrary bipartite graph (c) $K_{3,3}$ (d) W_6 (e) An arbitrary distance-hereditary graph (f) Petersen graph (g) Domino graph (h) Twisted Domino drawing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

2.3 Graph classes

Complete graphs In a complete graph for each vertex v_i it holds that there exists edges $\{(v_i, v_j) | \forall j \in \{1..n\} \setminus \{i\}\}$. A complete graph with n vertices is denoted with K_n. For complete graphs we already know, that they have strict outerconfluent drawings. In Figure 2.1(a) a K_6 can be seen.

Bipartite graphs In a bipartite graph the set of vertices V can be split into subsets V_1, V_2 such that $V = V_1 \cup V_2 \wedge V_1 \cap V_2 = \emptyset$ such that there does not exist an edge (v_i, v_j) such that $(v_i \in V_1 \wedge v_j \in V_1) \vee (v_i \in V_2 \wedge v_j \in V_2)$. In Figure 2.1(b) an arbitrary bipartite graph is depicted.

Complete bipartite graphs A complete bipartite graph is a bipartite graph in which it also holds that for each vertex $v_i \in V_1$ there exist edges (v_i, v_j) such that $\forall v_j \in V_2$ and for each vertex $v_i \in V_2$ there exist edges (v_i, v_j) such that $\forall v_j \in V_1$. $K_{m,n}$ therefore denotes a complete bipartite graph with m and n being the number of elements in each set. For example $K_{3,4}$ is a complete bipartite graph where $|V_1| = 3$ and $|V_2| = 4$. The graph $K_{3,3}$ for is one of the Kuratowski minors without an outerplanar drawing. For the class of complete bipartite graphs we already know that they have strict outerconfluent drawings. Figure 2.1(c) shows a $K_{3,3}$.

Wheels A wheel is formed by first creating a cycle of n vertices and then adding one vertex which is connected to each other vertex except itself. W_n denotes a wheel with a cycle of size n. For wheels with $n \geq 4$ it is known that they do not obtain an outerplanar drawing. Wheels with $n \geq 5$ do not obtain strict outerconfluent drawings. In Figure 2.1(d) the wheel W_6 can be found.

Distance-hereditary graphs A distance-hereditary graph is a graph constructed from a single vertex by applying a sequence of the following three operations:

Adding a pendant vertex: Add a new vertex u to the graph that is adjacent to exactly one other vertex v of the graph.
Creating a false twin: For an existing vertex v add a new vertex u with exactly the same neighbours as v.
Creating a true twin: For an existing v create add a false twin u and add (u, v).

All three operation are depicted in Figure 2.2. By construction we know that distance-hereditary graphs admit strict outerconfluent drawings. In Figure 2.1(e) an arbitrary distance-hereditary graph is shown.

Petersen graph The Petersen graph is the graph depicted in Figure 2.1(f). Since the Petersen graph does not have a confluent drawing, we also know that the graph does not have strict outerconfluent drawing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

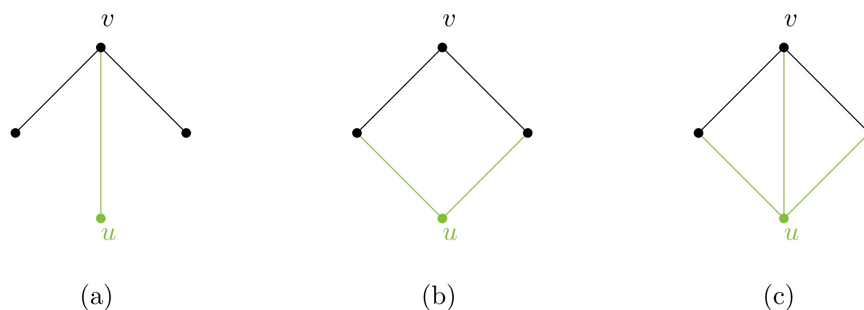


Figure 2.2: Operations of distance-hereditary graphs: (a) Adding a pendant vertex (b) Creating a false twin (c) Creating a true twin

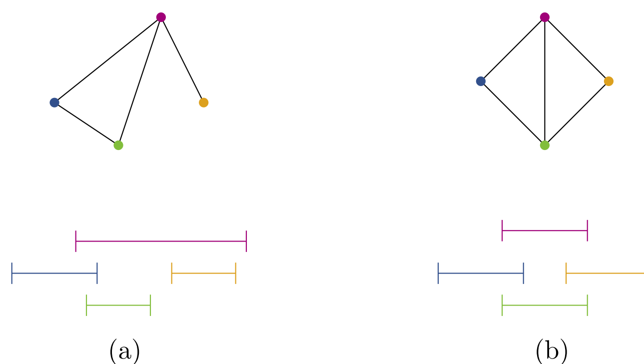


Figure 2.3: (a) Construction of an interval graph (b) Construction of an unit-interval graph

Domino graph The Domino graph is the graph pictured in Figure 2.1(g). We construct this graph by creating two cycle of size four and merge two connected vertices of the one cycle with two connected vertices of the other cycle.

Twisted Domino The Twisted Domino is a certain drawing of the Domino graph seen in Figure 2.1(h).

Interval graph An interval graph is a graph, having an intersection model consisting of intervals on a straight line. In other words we place intervals of different size on a line. Then we represent each interval as vertex in the graph and if two intervals intersect we create an edge between the corresponding vertices. We do not know whether interval graph obtain strict outerconfluent drawing or not. An example of how a interval graph can be constructed can be seen in Figure 2.3(a).

Unit-interval graph An unit-interval graph is an interval graph for which the intervals all have the same size. Therefore unit-interval graphs are a subset of intervals graphs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

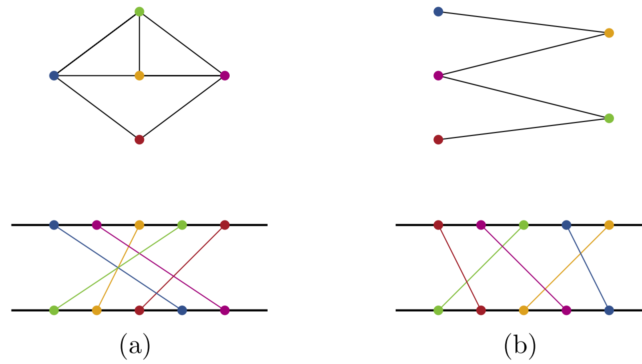


Figure 2.4: (a) Construction of a permutation graph (b) Construction of a bipartite-permutation graph

As for interval graphs we do not know if unit-interval graphs obtain strict outerconfluent drawings. In Figure 2.3(b) the construction of an unit-interval graph is pictured.

Permutation graph A permutation graph is a graph, having an intersection model consisting of straight lines between two parallels. In other word we create two parallels and place the same number of nodes on both of the lines. We then create straight lines between two nodes (one of each parallel). To construct the graph we create a vertex for each straight line and an edge between two vertices if the corresponding lines intersect. It is unknown whether permutation graphs have strict outerconfluent drawings or not. In Figure 2.4(a) the construction of a permutation graph is shown.

Bipartite-permutation A bipartite-permutation graph is a permutation graph which is also a bipartite graph. Hence bipartite-permutation graphs are a subset of permutation graphs. Furthermore we do not know if this class obtains strict outerconfluent drawings. Figure 2.4(b) show the creation of a bipartite-permutation graph.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Description of the algorithm

In this chapter we describe the algorithm in detail. First, a short description of the algorithm is given and then for each individual step further details are given in a dedicated subsection.

The algorithm gets a graph G with an ordered set of vertices as input. We assume the vertices are placed on the boundary of a topological disk. Let π be the inferred clockwise circular order of the vertices. Furthermore, assume that A is the adjacency matrix of G with rows and columns ordered according to π. The output of the algorithm is a strict outerconfluent drawing Γ, if it exists. If there is no strict outerconfluent drawing Γ for the given graph, the algorithm returns *FALSE*.

Algorithm 3.1: SOC-Algorithm

Input: A graph $G = (V, E)$, a vertex ordering π
Output: The strict-outerconfluent drawing Γ
1 $T \leftarrow$ createTableT(G);
2 $N^{\circlearrowright} \leftarrow$ createClockwiseTableN(G);
3 $N^{\circlearrowleft} \leftarrow$ createCounterclockwiseTableN(G);
4 *funnels* $\leftarrow$ findFunnels(G,$N^{\circlearrowright}$,$N^{\circlearrowleft}$) ; `/* might abort */`
5 *JS* $\leftarrow$ createJunctionSkeleton(G,*funnels*) ; `/* might abort */`
6 *CD* $\leftarrow$ createCanonicalDiagram(*JS*,T,G) ; `/* might abort */`
7 $\Gamma \leftarrow$ createSOCDrawing(*CD*);
8 **return** Γ

Algorithm 3.1 gives a short overview of the high-level steps of the algorithm. In Lines 1-3 data structures are computed, which are going to be used in later steps. Theses data structures are used to query certain properties of the graph in constant time. Afterwards, in Line 4 all funnels are found in the input graph. If the number of found funnels

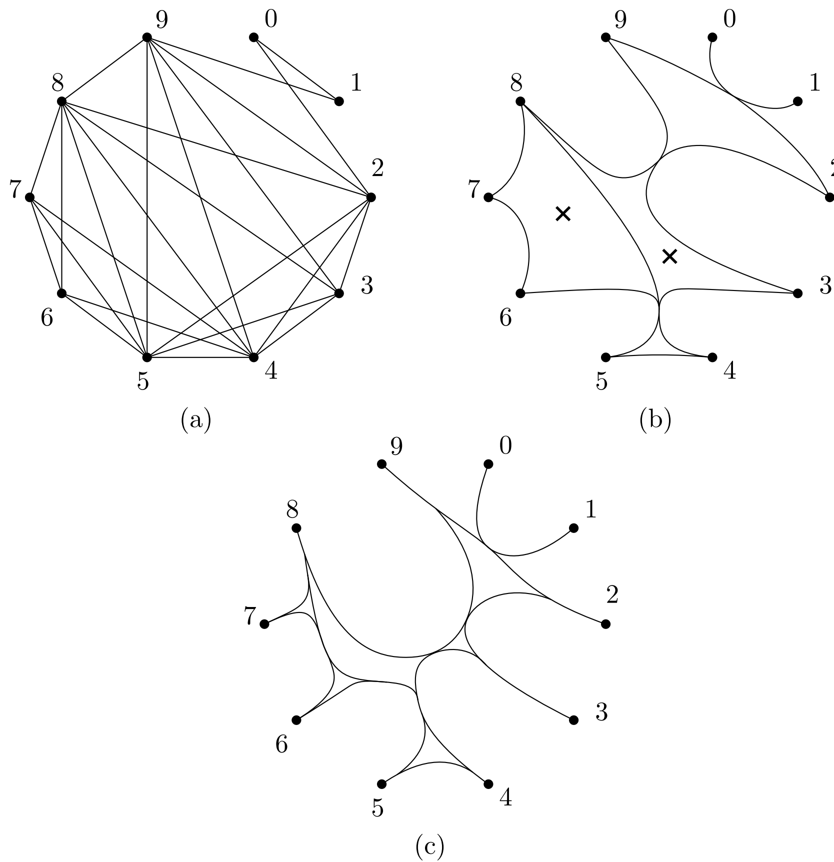


Figure 3.1: (a) Input graph G_E (b) Canonical diagram D_E of G_E (c) Strict outerconfluent drawing Γ_E of G_E

exceeds the upper bound in Lemma 1, the algorithm aborts at this step. Afterwards, the algorithm creates the so called junction skeleton in Line 5. If the created junction skeleton has no outerplanar drawing respecting π, the algorithm aborts. In Line 6 the canonical diagram is created by adding edges to the junction skeleton and marking the faces. If it is not possible to create a canonical diagram, the algorithm aborts at this step. Finally, in Line 7 the canonical diagram is used to create a strict outerconfluent drawing.

In the following sections detailed information about each step in Algorithm 3.1 is given. Throughout these sections an ongoing example is used to illustrate each step. The graph for this example is called *example graph* G_E and can be seen in Figure 3.1(a). The canonical diagram D_E as well as the strict outerconfluent drawing Γ_E of G_E can be seen in Figure 3.1(b) and (c).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.1 Lookup table T

Table T helps us counting the number of edges between two different intervals in constant time. We need T in later steps to determine whether there are edges between two disjoint intervals or not. Therefore we create table T in $O(n^2)$ time and then can query the needed information in $O(1)$ time.

Creation of T Table T is a summed area table for the adjacency matrix of the input graph G. More formally, $T(i,j) := |\{(v_{i'}, v_{j'})|i' \leq i, j' \leq j, (v_{i'}, v_{j'}) \in E\}|$. In Figure 3.2(a) the table lookup T_E for our example graph G_E (Figure 3.1 (a)) is displayed and in Figure 3.2(b) the adjacency matrix A_E of G_E can be seen. The coloured values in T_E are the sum of the corresponding rectangle in the adjacency matrix A_E.

It is also important to understand what the values of T represent. The value for $T[i,j]$ is the number of edges for which the source of the edge is between $[0,i]$ and the target of the edge is between $[0,j]$. In Figure 3.1(c) we give an example why the value for $T_E[8,6]$ is 20. The source interval $[0,8]$ is marked in darkblue and the target interval $[0,6]$ is marked in lightblue. The darkred coloured edges are part of both intervals. These edges need be counted twice, since both directions are considered. For the lightred edges it holds that one vertex is part of the source interval and the opposite vertex is part of the target interval, but not the other way around. Hence, these vertices will be counted once. Since we have eight vertices marked darkred and four vertices marked lightred we get $8 * 2 + 4 * 1 = 20$.

It remains to describe how table T can be calculated in an efficient way. The idea is to iterate the adjacency matrix exactly one time in order to have a running time of $O(n^2)$. This can be achieved by iterating over the rows of the A and building a sum of the current row while iterating. The values of the current row are then the values of the last row plus the sum of the current row up to this index. An exception is the first row, where the values are just the current sum of the row. The detailed code for this method is presented in Algorithm 3.2.

Usage of T As mentioned before T is used to determine the number of edges between two disjoint intervals $[a,b]$ and $[c,d]$ in a graph G. There are three different cases which are considered.

Case 1 Neither of the intervals contain both the first and the last index of the vertices. If this is the case, we assume w.l.o.g. that the interval $[a,b]$ is the first interval occurring in the clockwise vertex order and interval $[c,d]$ is the second interval. To calculate the number of edges between $[a,b]$ and $[c,d]$ we take the number of vertices starting between $[0,b]$ and ending between $[c,d]$. Using table T this is equal to $T[b,d] - T[b,c-1]$. From this number we now have to subtract the number of vertices starting between $[0,a-1]$ and ending between $[c,d]$. This number is just $T[a-1,d] - T[a-1,c-1]$.

Case 2 Now assume one if the intervals starts at the first vertex of the vertex ordering. W.l.o.g. assume this is interval $[a,b]$. To get the number of edges between $[a,b]$ and $[c,d]$

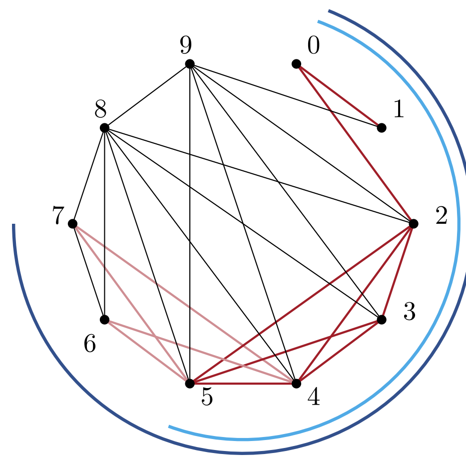
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	2	2	2	2	2	2	**2**
1	1	2	3	3	3	3	3	3	3	4
2	2	3	4	5	6	7	7	7	8	10
3	2	3	5	6	8	10	10	10	12	15
4	2	3	6	8	10	13	14	15	18	22
5	2	3	7	10	13	16	18	20	24	29
6	2	3	7	10	14	18	20	23	28	33
7	2	3	7	10	15	**20**	23	26	32	37
8	2	3	8	12	18	24	28	32	38	44
9	2	4	10	15	22	29	33	37	44	**50**

(a)

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	1
2	1	0	0	1	1	1	0	0	1	1
3	0	0	1	0	1	1	0	0	1	1
4	0	0	1	1	0	1	1	1	1	1
5	0	0	1	1	1	0	1	1	1	1
6	0	0	0	0	1	1	0	1	1	0
7	0	0	0	0	1	1	1	0	1	0
8	0	0	1	1	1	1	1	1	0	1
9	0	1	1	1	1	1	0	0	1	0

(b)



(c)

Figure 3.2: (a) Table T_E for graph G_E (b) Creating T_E by using the adjacency matrix of graph G (c) Calculation of $T_E[8,6]$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 3.2: createTableT(G)

Input: A graph $G = (V, E)$, a vertex ordering π
Output: The lookup table T

```
1  n ← numberOfVertices(G) ;
2  T ← int[n,n];
3  A ← getAdjacencyMatrix(G) ;
4  for i ← 0 to n − 1 do
5  |   for j ← 0 to n − 1 do
6  |   |   rowSum ← rowSum + A[i, j] ;
7  |   |   if i = 0 then
8  |   |   |   T[i][j] ← rowSum;
9  |   |   else
10 |   |   |   T[i][j] ← rowSum + T[i − 1][j];
11 |   |   end
12 |   end
13 end
14 return T
```

we calculate the number of edges starting between $[0, b]$ ending between $[c, d]$. This can be done by first accessing $T[b, d]$ and then subtract $T[b, c-1]$ from it.

Case 3 For this case it holds that one of the intervals, includes the first as well as the last vertex in the ordering π. W.l.o.g. assume $[a, b]$ is this interval. In order to calculate the number of edges between $[a, b]$ and $[c, d]$ we use Case 1 as well as Case 2. We split interval $[a, b]$ into the intervals $[a, n-1]$ (n being the number of vertices) and $[0, b]$. For the number of edges between interval $[0, b]$ and $[c, d]$ we use Case 2 and between interval $[c, d]$ and $[a, n-1]$ we use Case 1. Adding both values results in the total number of edges between those intervals.

Following functions f defines how T is used to calculate the number of edges between two disjoint intervals. The order of the cases in the function is equal to the cases defined above.

$$f([a, b], [c, d]) = \begin{cases} (T[b, d] - T[b, c-1]) & \\ -(T[a-1, d] - T[a-1, c-1]) & 0 < a < b < c < d < n \\ & \\ (T[b, d] - T[b, c-1]) & 0 = a < b < c < d < n \\ & \\ (T[b, d] - T[b, c-1]) & \\ +(T[d, n-1] - T[d, a-1]) & \\ -(T[c-1, n-1] - T[c-1, a-1]) & 0 \leq b < c < d < a < n \end{cases}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

3.2 Lookup tables N

The lookup tables $N^{\circlearrowright}$ and $N^{\circlearrowleft}$ are data structure which will be used in later steps of the algorithm. They are used to find the next neighbour of a vertex v clockwise/counter-clockwise of a vertex w. Since this information is queried often in later steps, we want to create the tables $N^{\circlearrowright}$ and $N^{\circlearrowleft}$ once in $O(n^2)$ time. Afterwards the information is accessed in $O(1)$ time.

Creation of $N^{\circlearrowright}$ The value of $N^{\circlearrowright}[v, w]$ is the index of vertex u such that u is adjacent to v and there is no vertex u' adjacent to v and inside the interval $[w, u]$. If v is the same as w the value of the table is undefined, which does not matter since it is not used later on. Since we assume, that each vertex of G has at least one neighbour, there are no empty rows.

In Figure 3.3(a) the table $N_E^{\circlearrowright}$ for the graph G_E (Figure 3.1(a)) can be found. Considering the value of $N_E^{\circlearrowright}[5, 2]$ (marked in green) is 3. This means the next neighbour of vertex 5, clockwise of vertex 2, is vertex 3. In Figure 3.3(c) we can check that this is indeed the case. In order to compute the value of $N^{\circlearrowright}[v, w]$ we could simply walk from vertex w along the clockwise order until we find the first neighbour of v. Doing so for each ordered pair of indices would be poly-time in the number of vertices, but would break our desired running time bound of $N(O^2)$. Fortunately there is a way of calculating $N^{\circlearrowright}$ in $N(O^2)$.

We again calculate row after row and will access each row of the adjacency matrix at most two times. This can be done by performing a counterclockwise scan over π. For each row i we start at the next vertex counterclockwise of the vertex with index i. Then we iterate through all vertices in counterclockwise order until $N^{\circlearrowright}[i, \cdot]$ except $N^{\circlearrowright}[i, i]$ has values set. The index k is the index of last found neighbour of i. For each index j we first set the value of $N^{\circlearrowright}[i, j]$ to k if k has been set yet. Then if j is also a neighbour of i we set k to j.

In Algorithm 3.3 the detailed pseudo code for calculating table $N^{\circlearrowright}$ in an efficient way is presented. There, we variable, called *lastNeighbour*, to save the last found neighbour and we use another variable called *firstIndexSet* which simply holds the index of the first value set for the current row. Given the pseudo code it is clear that this algorithm runs in $O(n^2)$.

Creation of table N counterclockwise The value of $N^{\circlearrowleft}[v, w]$ is the index of vertex u such that u is adjacent to v and there is no vertex u' adjacent to v and inside the interval $[u, w]$. For $N^{\circlearrowleft}$ it also holds that if v is the same as w the value of the table is undefined and if there is a vertex without a single neighbour the according row is undefined.

The principles of calculating the values $N^{\circlearrowleft}$ are almost the same as for $N^{\circlearrowright}$. In Figure 3.3(b) table $N^{\circlearrowleft}$ for the example graph G (Figure 3.1(a)) can be seen. The value of $N^{\circlearrowleft}[5, 2]$ (marked in red) is 9 which means that next neighbour of 5 counterclockwise of

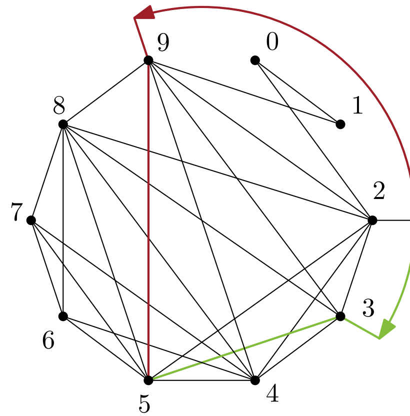
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

–	2	1	1	1	1	1	1	1	1
9	–	9	9	9	9	9	9	9	0
3	3	–	4	5	8	8	8	9	0
2	2	4	–	5	8	8	8	9	2
2	2	3	5	–	6	7	8	9	2
2	2	**3**	4	6	–	7	8	9	2
4	4	4	4	5	7	–	8	4	4
4	4	4	4	5	6	8	–	4	4
2	2	3	4	5	6	7	9	–	2
1	2	3	4	5	8	8	8	1	–

(a)

–	2	1	2	2	2	2	2	2	2
9	–	0	0	0	0	0	0	0	0
9	0	–	0	3	4	5	5	5	8
9	9	9	–	2	4	5	5	5	8
9	9	9	2	–	3	5	6	7	8
9	9	**9**	2	3	–	4	6	7	8
8	8	8	8	8	4	–	5	7	8
8	8	8	8	8	4	5	–	6	8
9	9	9	2	3	4	5	6	–	7
8	8	1	2	3	4	5	5	5	–

(b)



(c)

Figure 3.3: (a) Table $N^{\circlearrowleft}$ for graph G (b) Table $N^{\circlearrowright}$ for graph G (c) Calculation of $N^{\circlearrowleft}[5,2]$ and $N^{\circlearrowright}[5,2]$

2 should be 9. In Figure 3.3(c) it can be seen that this is the case. To create $N^{\circlearrowright}$ we use the same approach as for $N^{\circlearrowleft}$ but this time we do a clockwise scan. The procedure is the same, expect the direction we iterate trough the vertices. For the sake of completeness the pseudo code for creating $N^{\circlearrowright}$ can be found in Algorithm 3.4.

3.3 Discovery of the funnels

After creating data structures $T, N^{\circlearrowleft}$ and $N^{\circlearrowright}$ for G, the next step is to find all funnels of the graph. For every funnel we are going to create a junction which will be part of the canonical diagram and, consequently, the strict outerconfluent drawing. Furthermore, there will not be any further junctions than those which are created for the funnels in this step. Hence, the number of funnels found in this steps is the number of junctions in the final strict outerconfluent drawing. This step might abort (i.e. return FALSE) since the number of junctions in a strict outerconfluent is bounded by Lemma 1 and if the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 3.3: createClockwiseTableN(G)

```
   Input: A graph G = (V, E), a vertex ordering π
   Output: The table N↻
 1 n ← numberOfVertices(G) ;
 2 N↻ ← int[n,n];
 3 A ← getAdjacencyMatrix(G) ;
 4 for i ← 0 to n − 1 do
 5 |   lastNeighbour ← −1;
 6 |   firstIndexSet ← −1;
 7 |   j ← i − 1;
 8 |   while firstIndexSet ≠ j mod n do
 9 |   |   if not j mod n =  i then
10 |   |   |   if lastNeighbour ≠ -1 then
11 |   |   |   |   if firstIndexSet -1 then
12 |   |   |   |   |   firstIndexSet ← j;
13 |   |   |   |   end
14 |   |   |   |   N↻[i, j] ← lastNeighbour;
15 |   |   |   end
16 |   |   |   if A[i, j] =1 then
17 |   |   |   |   lastNeighbour ← (j mod n);
18 |   |   |   end
19 |   |   end
20 |   |   j ← j − 1;
21 |   end
22 end
23 return N↻
```

number of funnels we find is larger than this bound no strict outerconfluent drawing is possible for G with the given ordering of the vertices.

Calculating funnels To find the funnels we need to go through all separated intervals in the graph. For each separated interval $[a, b]$, vertex c is defined as the next neighbour of a that is counterclockwise of b, and vertex d is defined as the next neighbour of b that is clockwise of a. In order for these four vertices to form a funnel, following conditions must hold (see Eppstein et al. [EHL+16] page no. 40): (1) c is a neighbour of b, (2) d is a neighbour of a, (3) a is the next neighbour of c that is counterclockwise of b and (4) b is the next neighbour of d that is clockwise of c.

Since one possible condition for being a separated interval $[a, b]$ is that there exists a funnel $f = (c, d, e, f)$ inside $[a, b]$, we iterate over all possible interval sizes. First we check all intervals with size 1 and then increase the size by one and check again all intervals of this size. We repeat this until we checked all intervals up to size of $n - 1$ and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 3.4: createCounterclockwiseTableN(G)

```
   Input: A graph G = (V, E) with ordered vertices
   Output: The table N^↺
 1 n ← numberOfVertices(G) ;
 2 N^↺ ← int[n,n];
 3 A ← getAdjacencyMatrix(G) ;
 4 for i ← 0 to n − 1 do
 5 │ lastNeighbour ← −1;
 6 │ firstIndexSet ← −1;
 7 │ j ← i − 1;
 8 │ while firstIndexSet ≠ j mod n do
 9 │ │ if not j mod n =  i then
10 │ │ │ if lastNeighbour ≠ -1 then
11 │ │ │ │ if firstIndexSet -1 then
12 │ │ │ │ │ firstIndexSet ← j;
13 │ │ │ │ end
14 │ │ │ │ N^↺[i, j] ← lastNeighbour;
15 │ │ │ end
16 │ │ │ if A[i, j] =1 then
17 │ │ │ │ lastNeighbour ← (j mod n);
18 │ │ │ end
19 │ │ end
20 │ │ j ← j + 1;
21 │ end
22 end
23 return N^↺
```

as a result all funnels are found. In Figure 3.4(a) all funnels G_E can be seen, namely $f_1 = (0, 9, 2, 1)$(red), $f_2 = (2, 9, 8, 3)$(yellow) and $f_3 = (3, 6, 5, 4)$(green). Looking at funnel f_2, explicitly shown in Figure 3.4(b), we see that funnel f_1 is inside the interval $[a, b]$ of funnel f_2. Without funnel f_1 this interval would not be separated since there exists a vertex between a and b.

Algorithm 3.5 presents the pseudo code for finding all funnels in a given graph G. We use the look up tables $N^{\circlearrowleft}$ and $N^{\circlearrowright}$ computed for G. As mentioned earlier we iterate the interval size from 1 to $n - 1$ and for each interval size we check each possible interval of this size. For each interval $[a, b]$ we first find c (next neighbour of a counterclockwise of b) and d (next neighbour of b clockwise of a) by using $N^{\circlearrowleft}$ and $N^{\circlearrowright}$. Then we check if the interval $[a, b]$ is separated and if this is not the case we skip to the next interval. If the interval is separated we check conditions (1)-(4), defined earlier, which can be done in constant time using $N^{\circlearrowleft}$ and $N^{\circlearrowright}$. If all the conditions are fulfilled we add the funnel to our set of funnels F. Before returning all found funnels, we check if the number of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

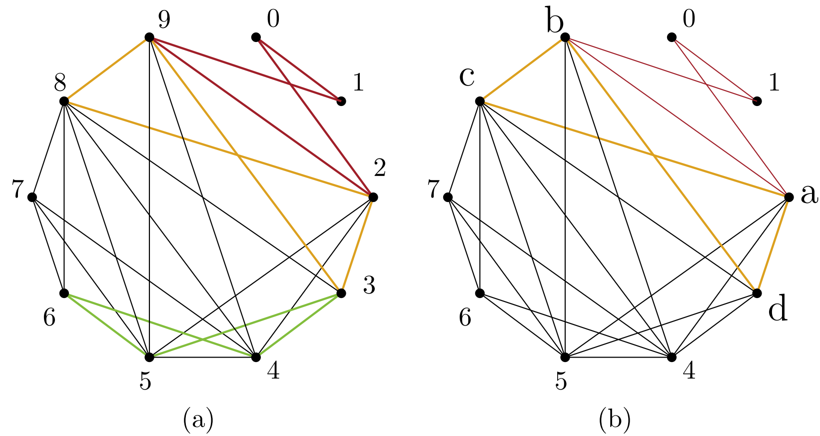


Figure 3.4: (a) Funnels of graph G_E (b) Funnel f_1 being inside funnel intervals of funnel f_2

funnels exceed the upper bound for junctions. If this is the case we return FALSE, which also will abort the base algorithm(Algorithm 3.1).

3.4 Creation of the junction skeleton

In this step we create the so called *junction skeleton* S. The junction skeleton contains all vertices and junctions as well as a subset of the edges of the canonical diagram. After the creation of the junction skeleton we check whether it has an outerplanar embedding respecting the given vertex order or not. We can do this in this step, since every following step of the algorithm only add uncrossed edges. If the junction skeleton has no such embedding the algorithm returns FALSE.

Building funnel trees Before building the junction skeleton we create a data structure called *funnel tree* T_v for each vertex v. All funnel trees will be stored in a map M with vertex v_i as key and funnel tree T_{v_i} as value. The funnel tree of a vertex represents the position of all funnels, whose funnel intervals include this vertex. The funnel tree therefore is an ordered tree with each node, except the root, having a funnel as label. The label of the root is denoted by $\emptyset$. The funnel tree stores clockwise order of edges per vertex as well as which junctions are needed to be connected.

To create the funnel tree T_v, we first gather all funnels where one of the funnel intervals starts or ends at this vertex. Each funnel f has two intervals $i_1 = [a, b]$ and $i_2 = [c, d]$. We first start by using the root as current root, and recursively apply different rules. In the following case distinction we always consider the interval not containing the vertex v when talking about an interval. The examples used for each case are not part of the example graph G_E, since the funnel trees of G_E are rather simple.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 3.5: findFunnels(G,$N^{\circlearrowleft}$,$N^{\circlearrowright}$)

Input: A graph $G = (V, E)$ with ordered vertices, Table $N^{\circlearrowleft}$, Table $N^{\circlearrowright}$
Output: Set of funnels

1 $n \leftarrow$ numberOfVertices(G) ;
2 $funnels \leftarrow$ Set<Funnel>;
3 $A \leftarrow$ getAdjacencyMatrix(G);
4 **for** $intervalSize \leftarrow 1$ **to** $n - 1$ **do**
5   **for** $a \leftarrow 0$ **to** $n - 1$ **do**
6         $b \leftarrow (a - intervalSize) \bmod n$;
7         $c \leftarrow N^{\circlearrowleft}[a][b]$;
8         $d \leftarrow N^{\circlearrowright}[b][a]$;
9   **if not** *isSeperatedInterval(a,b,funnels)* **or** $c = d$ **then**
10   continue;
11   **end**
12   **if** $A[b][c] = 1$ **and** $A[a][d] = 1$ **and** $N^{\circlearrowleft}[c][d] = a$ **and** $N^{\circlearrowright}[d][c] = b$ **then**
13   *funnels*.add(createFunnel(a,b,c,d));
14   **end**
15   **end**
16 **end**
17 **if** size(funnels) $> 3 * n - 6$ **then**
18   abort;
19 **end**
20 **return** *funnels*

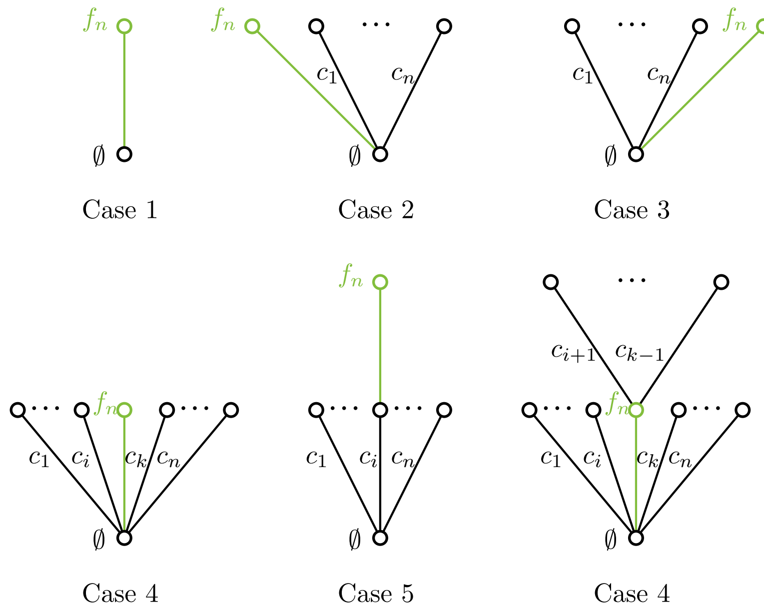


Figure 3.5: Cases 1-6 of inserting a new funnel f_n into an existing tree

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Case 1 The current root has no children yet. If this is the case the new node be will set as the only child of the current root. See Figure 3.5(Case 1) for an example.

Case 2 The interval of the new funnel is clockwise between v and the first child of the current root. In this case the new node will be set as the first child of the root. In Figure 3.5(Case 2) the root has children c_1 to c_n. Assume that the interval of the new funnel f_n. is between v and the interval of c_1s funnel. Therefore we add the new node to the left of c_1.

Case 3 The interval of the new funnel is clockwise between last child of the current root and v. In this case the new node will be added on the last place to the list of children. In Figure 3.5(Case 3) the root has children c_1 to c_n. Assume that the interval of the new funnel f_n is between the interval of c_ns funnel and v. According to this rule we add the new node right of c_n.

Case 4 The interval of the new funnel is clockwise between two children of the current root. In this case the new node is placed between those two children. In Figure 3.5(Case 4) we assume that the interval of the new funnel f_n is between the intervals of two children c_i and c_k. In this case the new node is added between c_i and c_k.

Case 5 The interval of the new funnel is inside one child of the current root. In this case the corresponding child will be set as the current root and the case distinction will be evaluated again with the new current root. Assume the root of the tree in Figure 3.5(Case 5) has one child c_i whose interval includes the interval of the new funnel f_n. Consequently, we take c_i as new root and evaluate this subtree again. Since c_i has not further children, we apply Case 1 and add the new node as child of c_i.

Case 6 The interval of the new funnel has a non-empty set of children of the current root inside it. In this case the set of children will be replaced by the new node and the new node has the set as his children now. In Figure 3.5(Case 6) we assume that the interval of the new funnel f_n includes the children c_{i+1} to c_{k-1} of the root. Therefore, all children between c_{i+1} and c_{k-1} will be set as children of our new node and the new node is placed between c_i and c_k.

Algorithm 3.6 shows the pseudo code for how a new funnel is inserted into an existing tree. This recursive algorithm is then used by Algorithm 3.7 whose purpose is to gather all funnels for a vertex and insert them into the tree one after another. In order to achieve this we create a map which has the vertices v_i as keys and the set of funnels f_i as values. This is achieved by iterating through each found funnel (a, b, c, d) and adding the funnel to the sets f_a, f_b, f_c and f_d . After creating this map we take the map value of each vertex v_i and start inserting one funnel after another to the funnel tree of this vertex. Finally we have the funnel tree map M we want for creating the junction skeleton S.

Creating the junction skeleton With M, we finally create the junction skeleton S. Recall that S consists of all vertices and junctions, and a subset of edges of the canonical diagram. The idea is to iterate through each of the funnel trees in M adding the needed edges according to the funnel trees.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

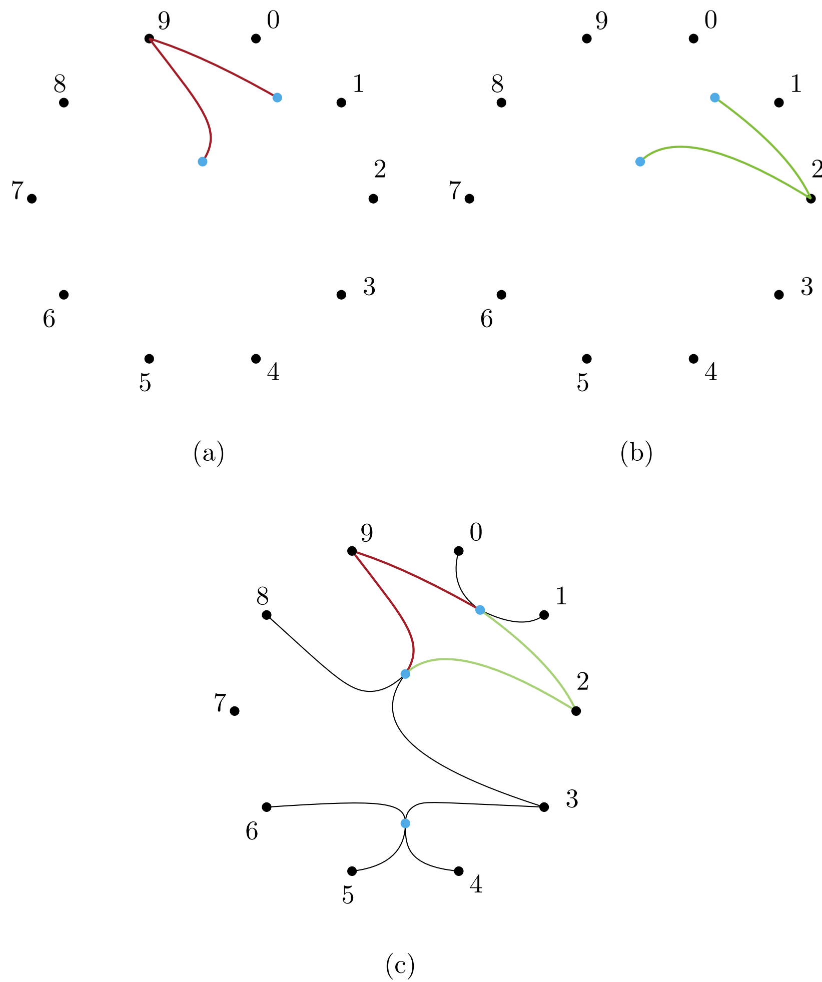


Figure 3.6: (a)Edges added while traversing through the funnel tree of vertex 9 (b)Edges added while traversing through the funnel tree of vertex 2 (c)Overlaying all added edges, resulting in the junction skeleton

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Algorithm 3.6: insertFunnelIntoTree(i,*funnel*,*funnelTree*)

Input: Index i of the vertex the funnel tree belongs to, Funnel *funnel* to insert into the tree, a node of the funneltree *funnelTreeNode*
Output: Funneltree with inserted funnel

```
1  children ← funnelTreeNode.getChildren();
2  if size(children)=0 then
3  |  children.add(new FunnelTreeNode(funnel));
4  |  return
5  end
6  included ← ∅;
7  funnelInterval ← funnel.getFunnel().getIntervalWithout(i);
8  foreach child in children do
9  |  childInterval ← child.getFunnel().getIntervalWithout(i);
10 |  if childInterval.includes(funnelInterval) then
11 |  |  insertFunnelIntoTree(i, funnel, child);
12 |  |  return
13 |  else if funnelInterval.includes(childInterval) then
14 |  |  included.add(child);
15 |  else if childInterval.isClockwise(i, funnel)) then
16 |  |  if size(included)>0 then
17 |  |  |  children.replace(included, new FunnelTreeNode(funnel, included));
18 |  |  else
19 |  |  |  children.addBefore(children, new FunnelTreeNode(funnel));
20 |  |  end
21 |  |  return
22 end
```

In a first step we create an empty graph and add all vertices of the original graph to it. Next for each vertex v_i we consider $M[v_i]$ and iterate through it using the depth-first search algorithm. When we reach one node t_i in the funnel tree we create a junction j_i for the funnel f_i labelled to the node, if this has not be done yet. Otherwise, we want to get j_i from our junction skeleton and add an edge between the j_i to the junction of the parent node of t_i. If the parent node of t_i is the root node, the edge will be created between v_i and j_i. The pseudo code for creating S for G is split into two parts. The setup algorithm, which introduces the data structure and adds all vertices, starts traversing through the trees is Algorithm 3.8. This algorithm also has the planarity check in it, which will be explained later in this section. Algorithm 3.9 implements the traversal through one tree and adds the necessary edges.

For G_E we exemplify this for vertices 9 and 2 in Figure 3.6(a) and (b). Going through all funnel trees, we get the junction skeleton seen in Figure 3.6(c). As shown in the figure, the junctions skeleton contains all vertices although some might not have appeared in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 3.7: createFunnelTrees(G,*funnels*)

```
   Input: A graph G = (V, E) with ordered vertices, Set of Funnels funnels
   Output: Map of Node to Funneltree
 1 n ← numberOfVertices(G) ;
 2 funnelTreeMap ← Map[int,FunnelTreeNode];
 3 funnelMap ← Map[int,Set[Funnel]];
 4 foreach funnel in funnels do
 5 |  funnelMap.get(funnel.getA()).add(funnel);
 6 |  funnelMap.get(funnel.getB()).add(funnel);
 7 |  funnelMap.get(funnel.getC()).add(funnel);
 8 |  funnelMap.get(funnel.getD()).add(funnel);
 9 end
10 for i ← 0 to n − 1 do
11 |  foreach funnel in funnelMap.get(i) do
12 |  |  funnelTreeMap.set(i,insertFunnelIntoTree(i,funnel, funnelTreeMap.get(i)));
13 |  end
14 end
15 return funnelTreeMap
```

Algorithm 3.8: createJunctionSkeleton(G,*funnels*)

```
   Input: A graph G = (V, E) with ordered vertices, Set of Funnels funnels
   Output: Junction skeleton junctionSkeleton
 1 junctionSkeleton ← new JunctionSkeleton() ;
 2 junctionSkeleton.setVertices(G.getVertices());
 3 n ← numberOfVertices(G) ;
 4 funnelTreeMap ← createFunnelTrees(G,funnels);
 5 for i ← 0 to n − 1 do
 6 |  foreach child in funnelTreeMap.get(i).getChildren do
 7 |  |  handleFunnelTreeNode(i,child,junctionSkeleton);
 8 |  end
 9 end
10 if not hasPlanarEmbedding(junctionSkeleton) then
11 |  abort;
12 end
13 return junctionSkeleton
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 3.9: handleFunnelTreeNode($sourceNode$,G,JS)

Input: Vertex in G we are coming from, node $funnelTreeNode$ of the funnel tree, Junction skeleton JS

```
1 junction ← JS.getOrCreateJunction(funnelTreeNode.getFunnel());
2 if not JS.hasEdge(junction, sourceNode) then
3 |  JS.createEdge(junction, soureNode);
4 end
5 foreach child in funnelTreeNode.getChildren do
6 |  handleFunnelTreeNode(junction,child,JS);
7 end
```

one of the funnel interval endpoints. We also can see that there exactly three junction (marked in blue), one for each funnel, which are also the junctions of the canonical diagram.

Since basically any data structure for an embedded planar graph works when storing the junction skeleton, we will not suggest a certain one. However, we need to assure that the junction skeleton has enough information such that a planar embedding test can be done. Therefore we need to prepare the data structure for this test during the creation. We stored the rotation system for each vertex and junction.

Checking the planarity Since we will not add anything ontop on S in later steps, such that the drawing would lose the property of having a planar embedding respecting π, we will perform a planarity check at this point. If S has no planar embedding, the algorithm will return FALSE. Since we use Euler's formula to determine whether it has a planar embedding or not, we have to calculate the number of faces. As mentioned in the step before, we stored a rotation system for each vertex and junction. With this rotation system we can compute the number of faces, by using the graph as a double connected edge list.

At this point, the junction skeleton S might not be a connected graph. In order to create the double connected edge list of S, we want to have a connected graph. Therefore, we add a so called outer cycle (Definition 1) O to S, resulting in a new graph S_O. We need to assure that S_O has a planar embedding if and only if S also has a planar embedding.

Definition 1 *(Outer Cycle) An outer cycle O for a junction skeleton S is the graph containing all outer vertices V of S and the set of edges between every two consecutive vertices v_i and v_{i+1} in the vertex order π. More formally $O = (V_O, E_O)$, where $V_O := V_S, E_O := \{(v_i, v_{i+1}) \mid v_i \in V_O \wedge 0 \leq i < n-1\} \cup \{(v_{n-1}, v_0)\}$*

Lemma 2 *The graph S_O created from S has a planar embedding respecting π, if and only if S has a planar embedding respecting π.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Proof. Let S be a junction skeleton and S_O created as above. We place all vertices corresponding to outer vertices of S on the boundary of a topological disk and, to junction on the inside of the disk. Therefore all edges of S have either one vertex on the boundary and the other vertex on the inside or both vertices on the inside. If S has a planar embedding then there are no two edges crossing each other. Each edge of O can be placed on the boundary of the topological disk. We also know that the edges of O can not intersect each other, since each edge does only go from one vertex on the boundary to the next vertex on the boundary.

Furthermore, the edges of S are placed such that they only intersect with the boundary of the topological disk at vertices. This means that the edges of S and the the edges of the O only meet each other at the vertices. Therefore S_O has no crossing and has a planar embedding.

If S has no planar embedding then for any drawing, there exist two edges e_1 and e_2 crossing each other. When creating S_O we only add additional edges, which can not remove the non-planar structure in S. Therefore, e_1 and e_2 are still crossing and S_O has no planar embedding. □

Lemma 3 *S_O is connected.*

Proof. Observe that the subgraph of S_O induced by the outer vertices is a connected cycle. Hence, if all the junctions vertices are connected to at least one outer vertex S_O is connected. In the construction of S, each junction is part of a funnel tree and each funnel tree has an outer vertex as its root. □

After creating S_O we want to calculate the number of faces of S_O. Therefore we first create a list L of directed edges. For each undirected edge $e = (v_1, v_2)$ we create two directed edges $e_1 = (v_1, v_2)$ and $e_2 = (v_2, v_1)$ and add them to L. After creating L we iterate through the edges in L. For every edge $e = (v_1, v_2)$ we start at vertex v_2 and get the next neighbour v_3 of v_2 counterclockwise of v_1. We delete (v_2, v_3) from L. Then, we again search for the the next neighbour v_4 of v_3 counterclockwise of v_2. We afterwards remove (v_3, v_4) from L. We continue these two steps until we reach v_1 again. When reaching the starting vertex v_1 we increase the number of faces by one. When doing this until L is empty, the result is the number of faces of S_O. Finally we have to check if Euler's formula is fulfilled or not. The pseudo code for the planarity check is described in Algorithm 3.10.

In Figure 3.7 the junction skeleton with outer cycle S_{O_E} of the graph G_E is shown. The red edges are the outer cycle O_E we added. Furthermore all faces were calculated as described before, which can be seen by the sketched cycle we discovered for each face. G_E has 11 faces, 13 vertices and 22 edges, and therefore satisfies Euler's formula 13 - 22 + 11 = 2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

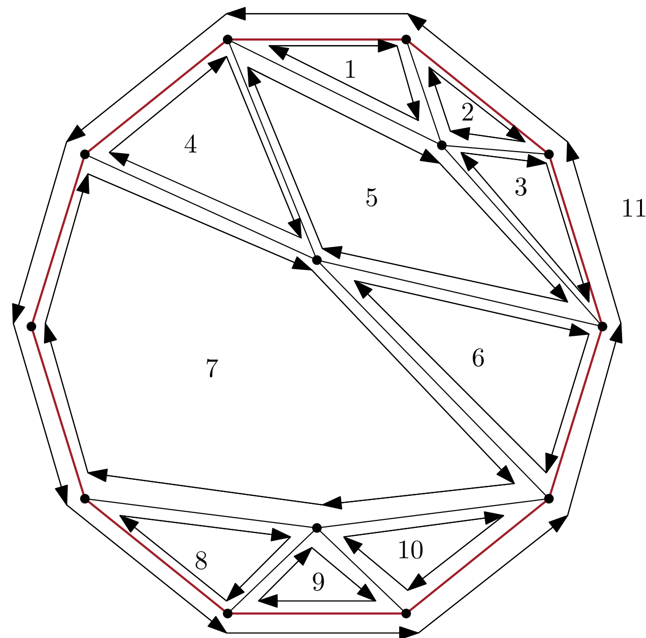


Figure 3.7: S_{O_E} with all found faces

3.5 Creation of the canonical diagram

So far we have created the junction skeleton S, containing all vertices and junctions of the canonical diagram D and a subset of the edges. Reaching this point we know that the junction skeleton S has a planar embedding respecting the vertex order π. At this point we need to add the remaining edges to the junction skeleton and mark certain faces. This will result in a graph which might be a canonical diagram for the input graph. This is tested by checking if exactly the edges of the original graph are represented by paths. Furthermore, there might be multiple paths between different vertices in this graph. So we also need to be sure that the graph is strict. If one of these conditions is violated the algorithm returns FALSE.

Create the canonical diagram In order to create the canonical diagram D from G, we first create a copy of the junction skeleton S. Then we add an outer cycle O to the junction skeleton as described in the section before. Afterwards we add the edges of the outer cycle which are in the original graph and not represented in the junction skeleton. Next we get all faces, expect the outer face, of S_O and handle each face separately.

For each face f we want to insert the missing edges that have to pass through this face. To find all these edges we take each node in the face and check which nodes in the face, expect itself and its neighbours, need to be connected to this node. If two nodes need to be connected inside the face, it means that there is a path in the original graph which has not be included yet. We usually distinguish between four cases that might appear:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 3.10: hasPlanarEmbedding(G,*funnels*)

```
   Input: Junction Skeleton junctionSkeleton
   Output: TRUE/FALSE if it has a/no planar
1  edgeList ← new List();
2  jsWithCycle ← junctionSkeleton.addOuterCycle();
3  faceCount ← 0 ;
4  foreach edge in jsWithCycle do
5  |   edgeList.add(edge.getSource(), edge.getTarget());
6  |   edgeList.add(edge.getTarget(), edge.getSource());
7  end
8  foreach edge in edgeList do
9  |   currentVertex ← edge.getSource();
10 |   lastVertex ← edge.getTarget();
11 |   while edge.getSource() != currentVertex do
12 |   |   newCurrentVertex ←
   |   |    currentVertex.getCounterClockwiseVertex(lastVertex);
13 |   |   lastVertex ← currentVertex;
14 |   |   currentVertex ← newCurrentVertex;
15 |   |   edgelist.remove(lastVertex, currentVertex);
16 |   end
17 |   faceCount ← faceCount + 1;
18 end
19 if jsWithCycle.getVertexCount() - jsWithCycle.getEdgeCount() + faceCount = 2
    then
20 |   return True
21 end
22 return False
```

Case 1 If both nodes represent vertices of the junction skeleton, then we use the adjacency matrix of G to determine whether there is an edge between those vertices or not.

Case 2 In this case one of the nodes is a vertex S and the other one is a junction in S, having one side into the face. That means we have to check whether there is an edge between the vertex and the other side of the junction (the side not inside the face). This interval goes from the most counter-clockwise to the most clockwise path leaving the other side of the junction. We use the lookup table T to calculate the number of edges needed between those intervals. If the number is greater than zero, an edge is needed.

Case 3 Both nodes are junctions of the junction skeleton, having one side facing towards the face. If this is the case we calculate the number of vertices between the junction sides, not facing the considered face, by using lookup table T. If this number is greater then zero, there has to be an edge between both junctions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

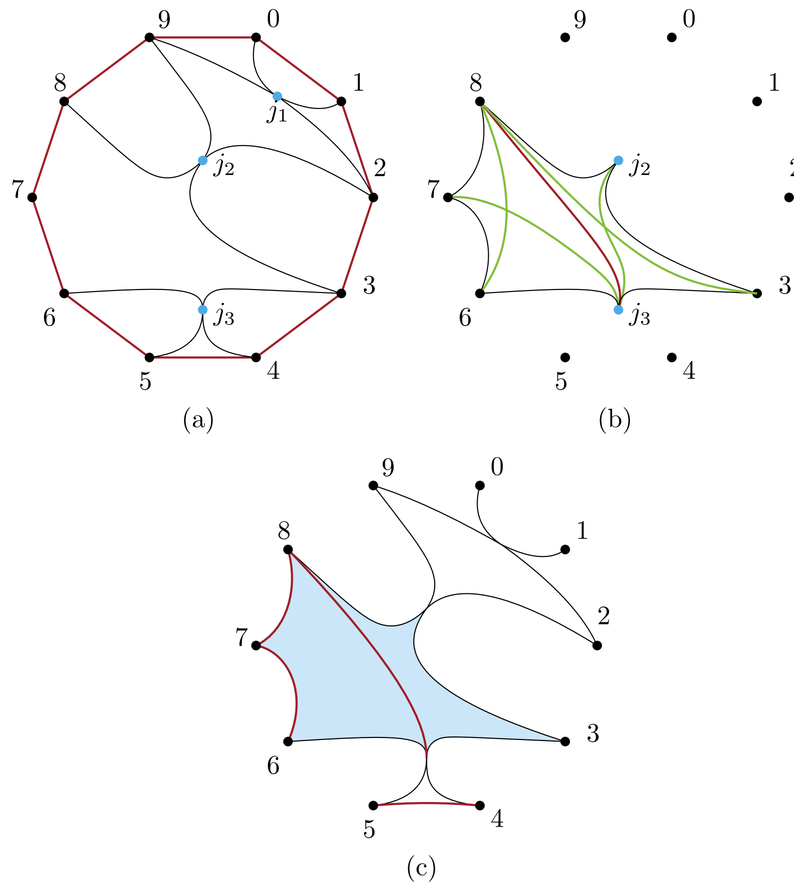


Figure 3.8: (a)S_{O_E} of graph G_E (b)Face $f = (j_2, 3, j_3, 6, 7, 8)$ with one uncrossed and four crossed edges (c)D_E of G_E with two marked faces

Case 4 One of the nodes is a junction of the junction skeleton with no side towards the face. In this case there can not be an edge between the junction and the other vertex/junction. Therefore it can be skipped.

After having found all needed edges for one face f, we need to divide the set of needed edges into the set of uncrossed edges E_u and crossed edges E_c. This can be done by checking for each needed edge if it is crossed by any other needed edge inside the face or not. First all uncrossed edges will be added to the canonical diagram. Each uncrossed edge divides f into different subfaces $f_1 ... f_n$. For each subface f_i we want to determine whether it contains a crossed edge or not. If it contains a crossed edge and f_i is a face of the canonical diagram, we set f_i in our canonical diagram as a marked face. If the face contains crossed edges and is not face of the canonical diagram, we abort at this step, since theses edges will cross. The pseudo code for this procedure can be found in Algorithm 3.11. These steps will result in a possible canonical diagram, since we now also need to prove that the canonical diagram is strict and represents the original graph.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Algorithm 3.11: createCanonicalDiagram(*junctionSkeleton*, *T*,*G*)

```
Input: A graph G = (V, E) with ordered vertices, Junction skeleton
       junctionSkeleton, Lookup table T
Output: Canonical Diagram canonicalDiagram
1  canonicalDiagram ← junctionSkeleton;
2  junctionSkeleton ←junctionSkeleton.addOuterCycle();
3  canonicalDiagram.addNeedeEdgeOnOuterCycle(G, junctionSkeleton);
4  faces ← junctionSkeleton.getFaces();
5  faces.removeOuterFace();
6  foreach face in faces do
7      neededEdges ← getNeededEdges(face, junctionSkeleton, G. T);
8      uncrossedEdges ← getUncrossedEgdes(neededEdges);
9      crossedEdges ← getCrossedEgdes(neededEdges);
10     canonicalDiagram.addEdges(uncrossedEdges);
11     subFaces ← canonicalDiagram.getFacesInside(face);
12     foreach subFace in subFaces do
13         if subface.hasCrossedEdges(crossedEdges) then
14             if canonicalDiagram.hasFace(subface) then
15                 canonicalDiagram.addMarkedFace(subFace);
16             else
17                 return false;
18             end
19         end
20     end
21 end
22 if not isCorrectCanonicalDiagram(canonicalDiagram, G) then
23     return false;
24 end
25 return canonicalDiagram
```

In Figure 3.8(a) the junction skeleton with the added outer cycle S_{O_E} (marked in red) of graph G can be seen. The blue points represent the junctions j_1, j_2 and j_3, which we added in the step before. We now consider the largest face f with the nodes $(j_2, 3, j_3, 6, 7, 8)$ seen in Figure 3.8(b). In this face we have five needed edges $(8, j_3), (8, 6), (7, j_3), (j_2, j_3), (8, 3)$ which we split into one uncrossed edge (red) $(8, j_3)$ and four crossed edges (green) $(8, 6), (7, j_3), (j_2, j_3), (8, 3),$. The uncrossed edge now splits the face into two faces, f_1 with nodes $(j_2, 2, j_3, 8)$ and f_2 with nodes $(j_3, 6, 7, 8)$. Since both faces f_1 and f_2 are also faces of the canonical diagram and contain crossed edges, they are marked faces of the canonical diagram. Doing this for all faces of S_{O_E}, we get the possible canonical diagram seen in Figure 3.8(c). In this figure the new edges created in this step are marked in red, where as the marked faces are marked in blue.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Verification of the possible canonical diagram In the last step we created a possible canonical diagram for the input graph G. We now need to check that this possible canonical diagram is strict and contains the same paths as the original graph.

In order to check both conditions we use a modified depth-first search algorithm. For each vertex v of the canonical diagram we first find all other vertices which have a path between them in D. If this collection of vertices has duplicated entries or includes v we return FALSE, since this violates the strictness. Next, we verify that this collection is equal to the vertices reachable in the original graph. If this is not the case we return FALSE at this step, since the canonical diagram is not representing the original graph.

First we need to modify our canonical diagram in order to make it easier to compute the collection of reachable vertices. For all marked faces, we create edges between all nodes in that face. To get the collection of all reachable vertices of one vertex, we perform a modified depth first search. When we reach a junction we only propagate to the other side of the junction (the one we did not come from) and if we reach a outer vertex we add this vertex to the collection and do propagate further. It is also important, that junctions will not be added to the collection, as they are not part of the original graph.

Algorithm 3.12: isCorrectCanonicalDiagram(*canonicalDiagram*, G)

Input: A graph $G = (V, E)$ with ordered vertices, Canonical Diagram *canonicalDiagram*

Output: True if the canonical diagram represents graph G, False otherwise

```
1  canonicalDiagram.replaceMarkedFacesByEdges();
2  n ← numberOfVertices(G) ;
3  for i ← 0 to n − 1 do
4  |  reachableVerticesCD ← canonicalDiagram.getReachableVertices(i);
5  |  if reachableVerticesCD.hasDuplicates or reachableVerticesCD.contains(i)
   |   then
6  |  |  return false
7  |  end
8  |  reachableVerticesOG ← G.getReachableVertices(i);
9  |  if not reachableVerticesOG = reachableVerticesCD then
10 |  |  return false
11 |  end
12 end
13 return true
```

In Figure 3.9(a) the canonical diagram of graph G_E together with the reachable vertices of 0 (red marked) and 4 (blue marked) is depicted. At this point recall that it is possible to move along paths of marked faces although the angles are sharp. Therefore the paths of vertex 4 to 7, 9 and 2 go along marked faces. In Figure 3.9(b) the corresponding edges of the original graph are shown. It can be seen that vertex 4/0 can reach the same vertices in the canonical diagram as well as in the original graph.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

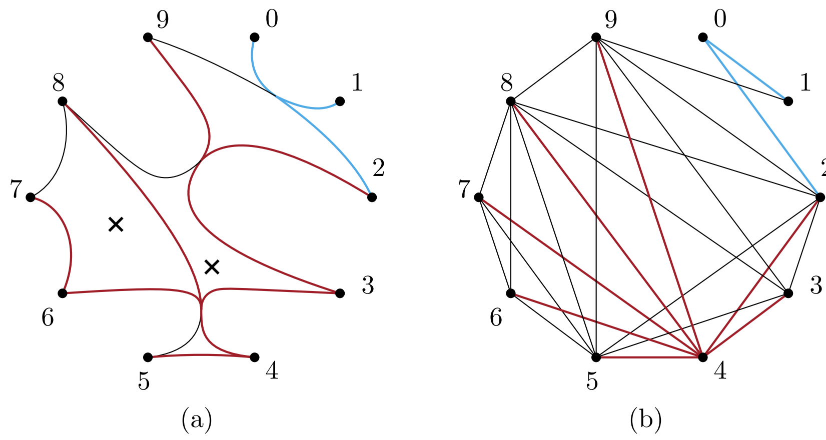


Figure 3.9: (a)Reachable vertices of vertex 4 and 0 in D_E (b)Reachable vertices of vertex 4 and 0 in G_E

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Implementation

In this chapter the implementation of the algorithm is described. First we explain what our implementation process looked like and then we give implementation specific details.

4.1 Implementation process

This section explains how the implementation of the algorithm described in chapter 3 was done. Since the algorithm is quite complex and has a lot of steps, it was necessary to be certain that all possible steps are developed correctly. To achieve this Test-Driven-Development was used, which means that for each step in the algorithm different tests were written before implementation was started. The main advantages of this approach is that we are able to constantly test all steps when modifying our implementation. This means that errors in existing code are found faster. It also assures that the implementation of new steps do not affect the implementation of already finished steps, since the tests show this immediately. In order to use Test-Driven-Development the algorithm needs to be deterministic. Furthermore, we need a huge test set including all possible cases per step.

Although we want our test set to cover all possible cases, we can not assure that we included them all. There might be some edge cases which are not considered or do not seem to be problematic. To prevent faults caused by missing edge cases in our tests set, we exhaustively test the finished implementation against certain graph classes. For these graph classes we already knew that every, one or no order has strict outer confluent drawing. Therefore we created all possible graphs of this class up to a certain number of vertices and tested each of them. Since we got the expected results for each of these graph classes, we are confident that our implementation is correct.

Type of test We use different kind of tests to verify our implementation. Each kind of test is automated and has a different purpose. At the end of the implementation all

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

tests must pass.

Unit tests Unit tests ensure that a module of the implementation is correct. In our application unit tests are used to verify a step of the algorithm. Therefore we have a set of unit tests for each step of the algorithm.

Integration tests Although all unit tests of the modules pass it can happen that the usage of different modules might fail. Integration tests prevent this, since they test whether combining different components works or not. Therefore integration tests test the whole system.

Exhaustive testing Since our unit tests can not cover all possible cases, we still want to be confident that our implementation is correct. Therefore we will test all possible graphs of certain graph classes up to a certain number of vertices. If the algorithm behaves correct for all instances this tests stage passes.

Process in details The implementation process is split into several tasks. A flow diagram showing the different tasks and their dependencies can be seen in Figure 4.1. We will now give a detailed description for this process.

First we want to understand each step of the algorithm in every detail. This is necessary since the algorithm is quite complex and the original description was vague.

Afterwards we create a test set. This set defines inputs and the expected output for each of the steps. The set contains arbitrary graphs as well as edge cases. Such an edge case is for example a graph which numbers of funnels is larger than the bound of Lemma 1. We use this test set to implement our tests.

In this step we write our automated unit and integration tests. Therefore we use the tests set created in the step before. After writing these tests, all of them should fail, since we did not implement any step yet. Since all our tests fails, we change the current state of our implementation. We change our implementation until all unit and integration tests pass.

After all unit and integration tests pass we start our exhaustive testing. If the algorithm does not behave like expected for a certain graph, we investigate why this is the case. Doing this we find out why our algorithm behaves incorrectly. Since we know the reason of failure we can add a unit test to our tests, testing this graph. Then we again start changing our implementation until all tests pass.

Graph classes for exhaustive testing We use different graph classes from whose we already know if they have a strict outer confluent drawing or not. There are three different kinds of graph classes. For the first kind it holds that each order π_i of the vertices has a strict outerconfluent drawing. To test theses classes we a created graph of this class and checked all possible $n!$ vertex orderings. If one π_i does not lead to a strict outer confluent drawing the test fails. A candidate of this class are complete graph, since each vertex ordering is isomorphic to the others.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

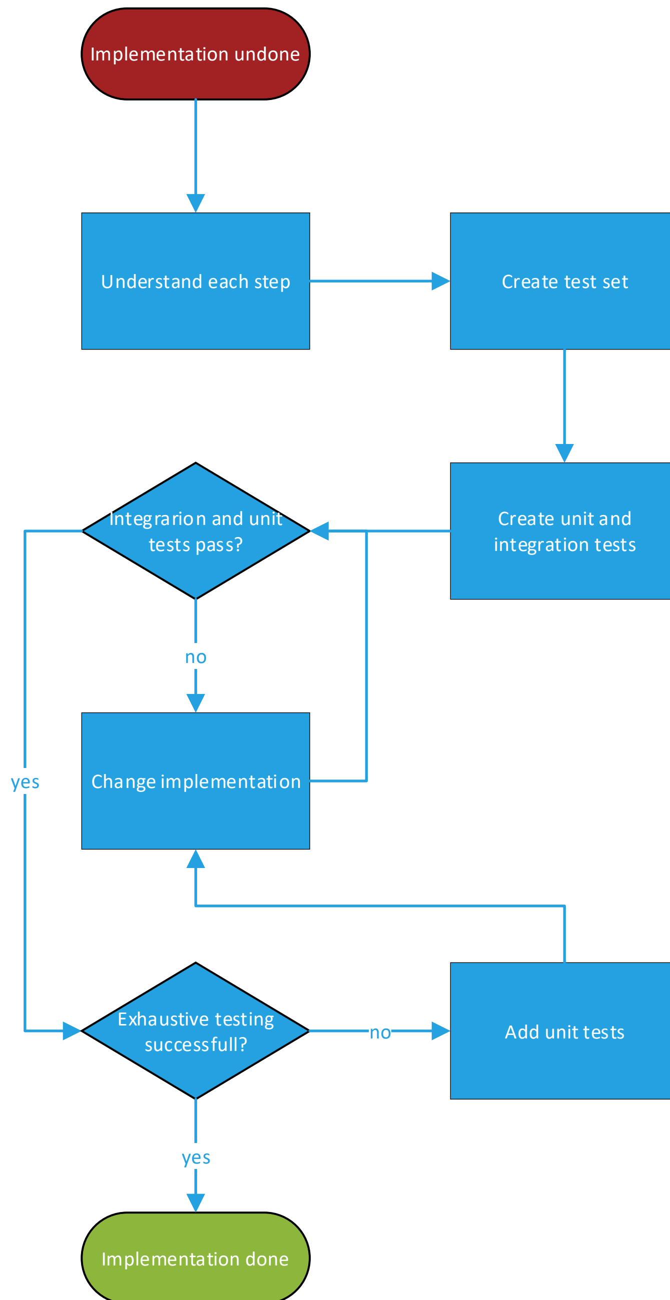


Figure 4.1: The used implementation process

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The second kind are graph classes having at least one vertex ordering π_i leading to a strict outer confluent drawing. To test these graph classes we go through all possible vertex orders until we find one positive instance. If there was no vertex order leading to a strict outer confluent drawing, then the test fails. The class of complete bipartite graphs is a representative for this this kind.

Last there are graph classes which do not have strict outerconfluent drawings. Therefore there exists no π leading to a drawing Γ. To test these classes of graphs if all vertex orderings π_i do not lead to a strict outer confluent drawing. However, if one vertex order leads to a strict outer confluent, then the test fails. We already know that class of wheels has this property.

In the following table all tested graph classes and there classification is shown:

All vertex orders	At least one vertex order	No vertex order
Complete graphs	Complete bipartite graphs	Wheels
	Distance-hereditary graphs	Petersen Graph
	Outerplanar graphs	

4.2 Implementation details

The implementation is done by using Java 11 together with JUnit 5 as testing library. The implementation expects the graph either as GraphML format or a text file with the ordered adjacency matrix as input. To define the vertex order when using a GraphML as input, we added a attribute to the node tag called position. The value of the position attribute is unique and has to be between 0 and $n-1$. If this is not the case the algorithm aborts with an error. The following XML code shows the input for $K_{3,3}$ with a correct vertex ordering. For sake of space we removed the attributes defining the source of the language in the *graphml* node.

```
<?xml version='1.0' encoding='utf-8'?>
<graphml>
        <graph edgedefault="undirected">
                <node id="0" position="5"/>
                <node id="1" position="0"/>
                <node id="2" position="4"/>
                <node id="3" position="1"/>
                <node id="4" position="3"/>
                <node id="5" position="2"/>
                <edge source="0" target="1"/>
                <edge source="0" target="3"/>
                <edge source="0" target="5"/>
                <edge source="1" target="2"/>
                <edge source="1" target="4"/>
                <edge source="2" target="3"/>
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
            <edge source="2" target="5"/>
            <edge source="3" target="4"/>
            <edge source="4" target="5"/>
        </graph>
</graphml>
```

If the graph has a strict outerconfluent drawing the algorithm returns a JSON file containing the data structure representing the canonical diagram. Otherwise, the algorithm prints a message saying that the graph has no strict outerconfluent drawing. The JSON file contains all vertices, edges, junctions and marked faces of the canonical diagram. For vertices and junctions the rotation system is also part of the output. Following JSON file shows the output for the test graph G_E.

```
{"nodes":[
        {"index":0,"rotation":[10]},
        {"index":1,"rotation":[10]},
        {"index":2,"rotation":[11,10]},
        {"index":3,"rotation":[12,11]},
        {"index":4,"rotation":[12,5]},
        {"index":5,"rotation":[12,4]},
        {"index":6,"rotation":[12,7]},
        {"index":7,"rotation":[6,8]},
        {"index":8,"rotation":[11,12,7]},
        {"index":9,"rotation":[10,11]}],
"junctions":[
        {"index":10,"rotation":[0,1,2,9],
                "side1":[0,9],"side2":[1,2]},
        {"index":11,"rotation":[2,3,8,9],
                "side1":[2,9],"side2":[3,8]},
        {"index":12,"rotation":[3,4,5,6,8],
                "side1":[3,6,8],"side2":[4,5]}],
"markedFaces":[
        {"nodes":[6,7,8,12]},
        {"nodes":[8,11,3,12]}]
}
```

Components We split the implementation of the algorithm into different components. For each step of the algorithm we created one component. Each component contains an interface as specification of the expected behaviour as well as several classes realising this specification. Each component, representing a step which might fail, has its own exception. This helps us identifying in which steps the algorithm returns FALSE. One advantage of this approach is that we can easily change the implementation of one step by just implement another implementation for the defined interface. Another advantage

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

is that we structure the code which helps when adding or changing code. As mentioned earlier we defined tests for each step of the algorithm. We have a test class for each component, performing unit tests for this component. If all unit tests for this component pass we know that the step represented by this component is correctly implemented.

Graph data structure As mentioned earlier every data structure for embedded planar graphs can represent the canonical diagram. However, we decided to use our own data structure which only contains the needed information. Since we developed the algorithm using Java, our data structure is object oriented. We use this data structure to represent the input graph, the junction skeleton as well as the canonical diagram. In Figure 4.2 the UML class diagram for our graph data structure is plotted.

A graph object has a list of edges, nodes (outer vertices) and junctions. Each of these objects (edges, nodes, junctions) cannot exist without a graph object. Therefore they are created by using public methods of the graph object. The attribute *id* of a node object is unique and identifies one node. A node furthermore can have a rotation, which has a list of neighbour nodes. A node does not always need a rotation because, we do not store the rotation for the input graph. A junction is a node with additional two sets of nodes representing the two sides of the junction. Since each side of the junction has at least two nodes, the cardinality is set to $2..n$. An edge is an object with two nodes as references and cannot exist if the two nodes do not exist.

A canonical diagram is a graph but also contains a set of faces. A face is am object that has a list of nodes, representing the clockwise order of nodes in the face. This list has at least four nodes, since a marked face has at least four vertices.

We always use private attributes for the data and define public methods for operating on this data. This should prevent a wrong usage of the data structure.

4.3 Problems during implementation

Graph data structure Deciding how our data structure representing the input graph, junctions skeleton and canonical diagram should look like was not trivial. We wanted to have a data structure capable to represent all necessary attributes of the classes as well as have all needed operation on this data but also to be as compact as possible. First we tried to use an already existing third party library called yFiles. At the first glance it seemed to be ideal for our usage, but when starting implementing we pretty soon found the limitations of this library. We hoped that the library is able to store the rotation system and perform a planarity check for the graph, but unfortunately this was not the case. Another problem using the library was that the classes used in this library did not have implemented the *equals* and *hashcode* methods. These methods are used to perform an equality check of two different objects and are used to perform the unit and integration tests. Since the effort to fix this problems would have been pretty high we then decided to implement our own data structure as described earlier this chapter.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

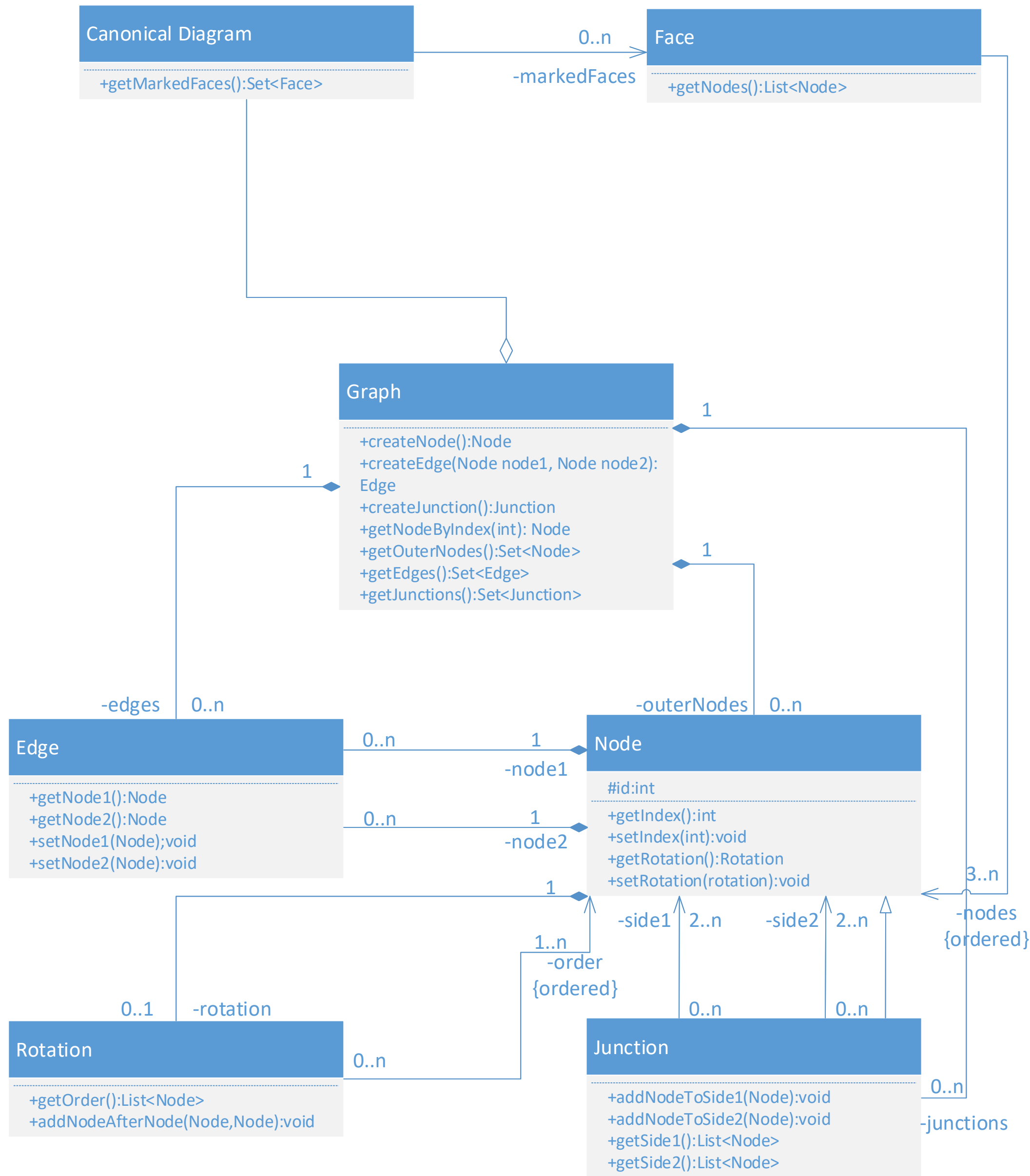


Figure 4.2: Class Diagram for our graph datastructure

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Object references We used objects to represent our input graph, junctions skeleton and canonical diagram. In some steps of the algorithm we for example have to add edges to the junction skeleton while still storing the junction skeleton as it was before. The problem in this case was that we could not simply clone the object representing the junction skeleton, since the objects of vertices, edges and junctions would have been the same as in original junction skeleton object. When we then add a new edge to the copied junction skeleton, the rotation of the vertices would change in both junction skeleton object, resulting in a wrong junction skeleton. To prevent this from happening we needed to create a so called *deep copy*. Therefore we create a new vertex, junction, rotation and edge object for each corresponding exiting object in our junction skeleton object.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek
WIEN Your knowledge hub

CHAPTER 5

Counterexample for bipartite permutation graphs

In this section we show that not all bipartite permutation graphs have strict outerconfluent drawings.

In [FGKN19] it was left as an open question if all bipartite permutation graphs admit a strict outerconfluent drawing. Our first intuition was that bipartite permutation graphs have strict outerconfluent drawings.

However, using the implementation of the algorithm we tested all possible vertex orderings for bipartite permutation graphs, which might not have an strict outerconfluent drawing, up to the size of 12 vertices. As a control program we wrote a python script which generated all bipartite permutations graphs for a specific number of vertices. Then the program calls the SOC-Algorithm implementation with a specific vertex ordering. If one graph has no vertex ordering that results in a strict outer confluent drawing we have a counter example. During this evaluation we a also measured the running time of our SOC-Algorithm implementation. Each call of the algorithm finished in less than one second, even if the input graph has 20 vertices. Therefore we think that the performance of our implementation is quite good. All our experiments were run on a standard desktop computer with an eight core Intel i7-6700 CPU clocked at 3.4GHz and 16GB RAM, running Archlinux with a current Linux kernel. Doing this experiments we found a counterexample for bipartite permutation graphs.

In the following theorem we will show that not every bipartite permutation graph admits a strict outerconfluent drawing by presenting a counterexample (graph P) which is shown in Figure 5.1(a). In Figure 5.1(b) it is depicted that P is indeed a bipartite graph and in 5.1(c) the permutation creating this graph is pictured. It remains to prove that P can not have a strict outerconfluent drawing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

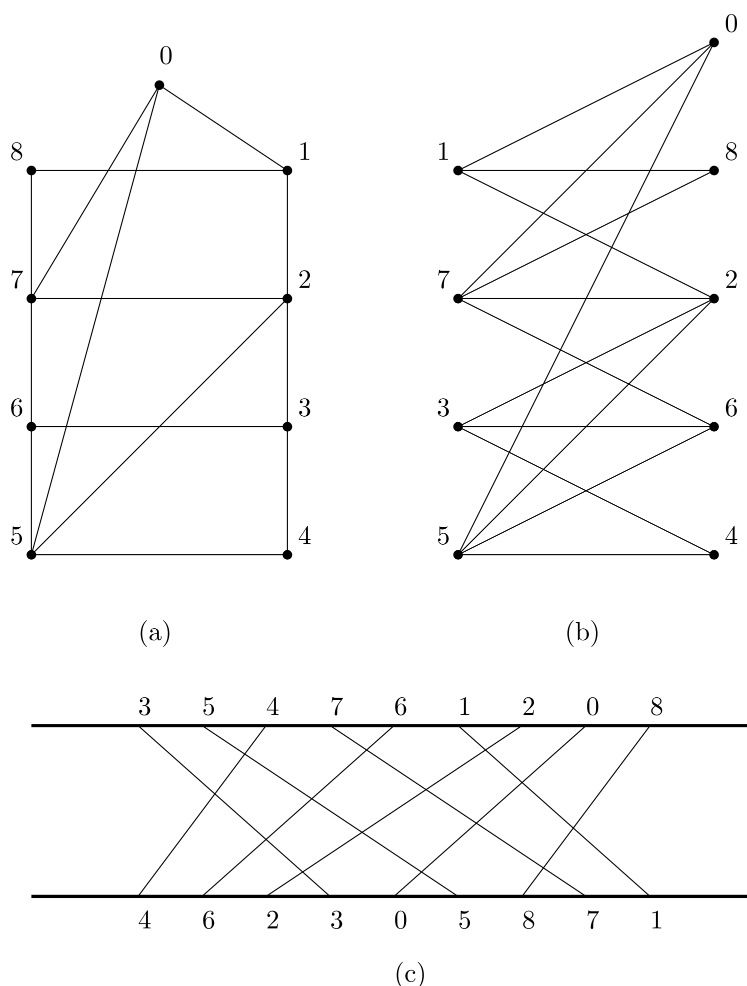


Figure 5.1: (a) Graph P (b) Drawing of P showing that P is bipartite (c) Permutation which generates P

Theorem 2 *There is a bipartite permutation graph that does not admit a strict outer-confluent drawing.*

Proof. First we consider drawings of the Domino graph, since graph P has Domino graphs as subgraphs. We want to know which vertex orders of the domino graph lead to an strict outerconfluent drawing. The Domino graph can be split into two cycles of length four. In Figure 5.2(a) the first cycle is marked in red and violet, the second cycle is marked in blue and violet. There can not be a strict outerconfluent drawing of P where a blue edge crosses a red edge. However the red and blue edges can cross the violet edge. Furthermore, all the vertices need to be on the boundary of an topological disk. Keeping this in mind the only valid drawings in terms of the crossing patterns are those seen in Figure 5.2(a)(b)(c) and the Twisted Domino. The Twisted Domino however has no strict

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

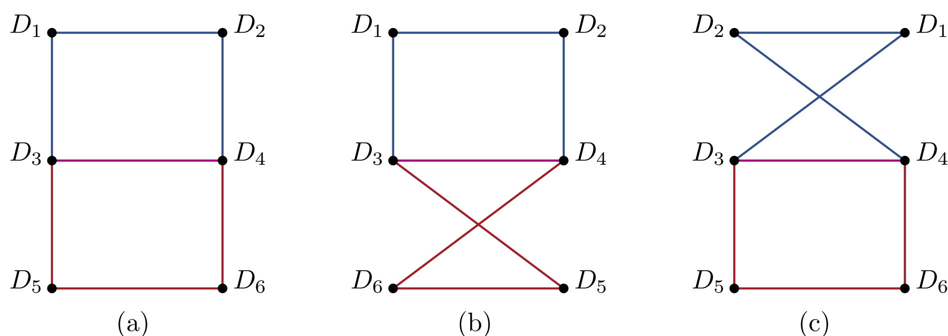


Figure 5.2: (a) Graph D (b)(c) Drawings of D

outerconfluent drawing and the graph drawing in Figure 5.2(b) and (c) are isomorphic. Hence, the drawings in Figure 5.2(a)(b) and the drawings isomorphic to them, are the only valid vertex orders for the Domino graph.

Now graph P has two Domino graphs as subgraphs where only one vertex is different, namely vertex 5 and 7. The subgraph consisting of both Domino graphs is called S and can be seen in Figure 5.3(a). The Domino subgraphs therefore are induced by the vertices $(0, 1, 2, 3, 5, 6)$ and $(0, 1, 2, 3, 6, 7)$. Now there are two non-isomorphic vertex orders $\pi_{P_1} = (1, 2, 3, 6, 5, 7, 0)$ and $\pi_{P_2} = (1, 2, 6, 3, 5, 7, 0)$ we can use to draw S.

Not twisting. In the first case we do not twist one of the side of the Domino graphs, pictured in Figure 5.3(a). Therefore we look at the strict outerconfluent drawing created by S in this vertex order. This drawing can be seen in Figure 5.3(b). It remains to add the vertices 4 and 8 to that drawing, which should result in a strict outerconfluent drawing. However there is no position in the vertex order to add 4 or 8 such that it would result in a strict outerconfluent drawing.

Twisting one end. In this case we twist one of the sides of the Domino graphs. The resulting vertex order can be seen in Figure 5.3(c). We again look at the strict outerconfluent drawing of S with this vertex order, which is depicted in Figure 5.3(d). Now we can place the vertex 4 such that a strict outerconfluent drawing can be created. The vertex 8 however can still not be placed in the vertex order such that a strict outerconfluent drawing exists.

Hence, it is impossible to add the vertices 4 and 8 to a vertex order, resulting in a strict outerconfluent drawing, for the structure S. Therefore, P can not have a strict outerconfluent drawing. □

However, graph P has a outerconfluent drawing when twisting both sides of the domino graphs in structure S. The outerconfluent drawing of P is pictured in Figure 5.3(f). This drawing is not strict, since there are multiple paths between the vertices 2 and 7 as well as 2 and 5.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

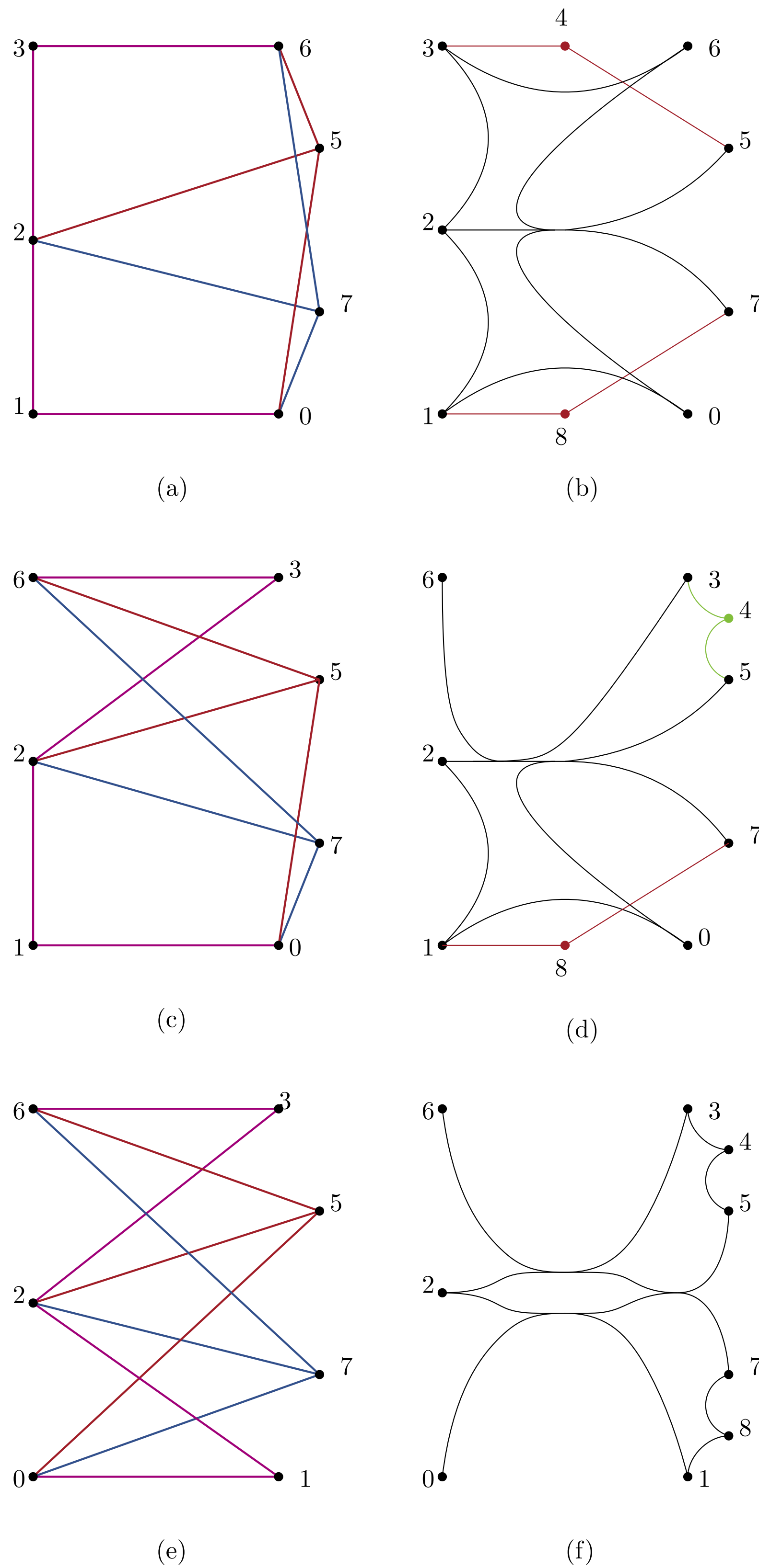


Figure 5.3: (a)Substructure S of P with vertex order π_{P_1} (b)Strict outerconfluent drawing of S with vertex order π_{P_1} (c)Substructure S of P with vertex order π_{P_2} (d)Strict outerconfluent drawing of S with vertex order π_{P_2} (e)Substructure S of P with vertex order π_{P_3} (f)Outerconfluent drawing of P

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

CHAPTER 6

Conclusion

In this thesis we implemented the algorithm to generate strict outerconfluent drawings from a given graph and vertex order described by Eppstein et al.[EHL+16]. We also gave a detailed explanation for each step supported by an ongoing example. We have also provided further insights into the development process as well as the implementation. Last but not least we evaluated different graph classes to find classes which do or do not obtain strict outerconfluent drawings. During this evaluation we found out that bipartite-permutation graphs do not obtain strict outerconfluent drawings. Therefore bipartite as well as permutation graphs do also not obtain these drawings.

Although we did a lot of work on the topic of strict outerconfluent drawings there are still some open problems. The most interesting question if the time needed to detect whether a given graph has a strict outerconfluent drawing or not is in polynomial time. One approach could be to find certain substructures in the graph which forbid or permit such a drawing.

Another open questions is if there are other graph classes like those we mentioned that do not or do admit strict outerconfluent drawings. We hope that our implementation helps further research towards this question, since it can for example be used for finding counterexamples.

Furthermore, it would be interesting to analyse the implementation itself and identify ways to improve its theoretical and practical running time. Since the original description of the algorithm was quite abstract it could be the case that certain steps were not implemented as efficient as it could be. Therefore a analysis of the running time of the whole implementation would be a good next step in this direction.

Last but not least it would be desirable to also create an actual drawing. This might not be a trivial step, since suitable curves must be identified and drawn. A simpler first step would be to use the circle packing approach by Eppstein et al. [EHL+16] or to draw the planar junction network.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

1.1 Confluent and strict outerconfluent drawings of K_5 and $K_{3,3}$ 2

2.1 Examples for defined graph classes 7
2.2 Operations of distance-hereditary graphs 9
2.3 Example of an interval graph 9
2.4 Example of an permutation graph 10

3.1 Input, canonical diagram and strict outerconfluent drawing of the *example graph* G_E 12
3.2 Table T_E of graph G_E and creations of table T_E 14
3.3 Tables $N^{\circlearrowleft}$ and $N^{\circlearrowright}$ of graph G, Creation of tables $N^{\circlearrowleft}$ and $N^{\circlearrowright}$ 17
3.4 Funnels of G_E and showing f_2 has separated funnel intervals 20
3.5 Examples of inserting a new funnel into a funnel tree 21
3.6 Explaining how the junctions skeleton is created 23
3.7 S_{O_E} with all found faces 28
3.8 Example of how the canonical diagram is created 30
3.9 (a)Reachable vertices of vertex 4 and 0 in D_E (b)Reachable vertices of vertex 4 and 0 in G_E 33

4.1 The used implementation process 37
4.2 Class Diagram for our graph datastructure 41

5.1 The counterexample for bipartite permutation graphs 44
5.2 Valid strict outerconfluent drawings of the domino graph 45
5.3 Explanation why P can not have a strict outerconfluent drawing 46

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

3.1 SOC-Algorithm 11

3.2 createTableT(G) 15

3.3 createClockwiseTableN(G) 18

3.4 createCounterclockwiseTableN(G) 19

3.5 findFunnels(G,$N^{\circlearrowright}$,$N^{\circlearrowleft}$) 21

3.6 insertFunnelIntoTree(i,*funnel*,*funnelTree*) 24

3.7 createFunnelTrees(G,*funnels*) 25

3.8 createJunctionSkeleton(G,*funnels*) 25

3.9 handleFunnelTreeNode($sourceNode$,G,JS 26

3.10 hasPlanarEmbedding(G,*funnels*) 29

3.11 createCanonicalDiagram($junctionSkeleton$, T,G) 31

3.12 isCorrectCanonicalDiagram(*canonicalDiagram*, G) 32

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[BRH+17] Benjamin Bach, Nathalie Henry Riche, Christophe Hurter, Kim Marriott, and Tim Dwyer. Towards unambiguous edge bundling: Investigating confluent drawings for network visualization. *IEEE Trans. Vis. Comput. Graph.*, 23(1):541–550, 2017.

[DEGM05] Matthew Dickerson, David Eppstein, Michael T. Goodrich, and Jeremy Yu Meng. Confluent drawings: Visualizing non-planar diagrams in a planar way. *J. Graph Algorithms Appl.*, 9(1):31–52, 2005.

[EGM05] David Eppstein, Michael T. Goodrich, and Jeremy Yu Meng. Delta-confluent drawings. In Patrick Healy and Nikola S. Nikolov, editors, *Graph Drawing, 13th International Symposium, GD 2005, Limerick, Ireland, September 12-14, 2005, Revised Papers*, volume 3843 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2005.

[EHL+16] David Eppstein, Danny Holten, Maarten Löffler, Martin Nöllenburg, Bettina Speckmann, and Kevin Verbeek. Strict confluent drawing. *Journal of Computational Geometry*, 7(1):22–46, 2016.

[FGKN19] Henry Förster, Robert Ganian, Fabian Klute, and Martin Nöllenburg. On strict (outer-)confluent graphs. In *27th International Symposium on Graph Drawing and Network Visualization (GD'19)*, 2019. To appear.

[HSS04] Peter Hui, Marcus Schaefer, and Daniel Stefankovic. Train tracks and confluent drawings. In János Pach, editor, *Graph Drawing, 12th International Symposium, GD 2004, New York, NY, USA, September 29 - October 2, 2004, Revised Selected Papers*, volume 3383 of *Lecture Notes in Computer Science*, pages 318–328. Springer, 2004.

[HvW09] Danny Holten and Jarke J. van Wijk. Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):983–990, 2009.

[PCJ97] Helen C. Purchase, Robert F. Cohen, and Murray I. James. An experimental study of the basis for graph drawing algorithms. *ACM Journal of Experimental Algorithmics*, 2:1–17, 1997.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Sch13] Marcus Schaefer. The graph crossing number and its variants: A survey. *The Electronic Journal of Combinatorics [electronic only]*, 20, 04 2013.

[ZPYQ13] H. Zhou, Panpan Xu, X. Yuan, and H. Qu. Edge bundling in information visualization. *Tsinghua Science and Technology*, 18(2):145–156, April 2013.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub