Technology
Arts Sciences
TH Köln

Technische Hochschule Köln
Faculty of Information, Media and Electrical Engineering

Master's Thesis

Generating Subfamilies of 1-Planar Graphs for automated Conjecture-Making

David Scholz

Cologne, February 2020

Master´s thesis

Fakultät für
Informations-, Medien-
und Elektrotechnik

Technology
Arts Sciences
TH Köln

Generating Subfamilies of 1-Planar Graphs for automated Conjecture-Making

Name: David Scholz

Student ID: 1 1 1 2 4 6 1 2

Program: Master Technische Informatik

Name 1st examiner: Prof. Dr. habil. Hubert Randerath

Name 2nd examiner: Dipl.-Math. Katharina Hammersen

Deadline: 0 3 . 0 2 . 2 0

Declaration:
I hereby confirm on my honour that this thesis is the result of my independent work.
All sources and auxiliary material used in this thesis are cited completely.

Place, date, signature: Cologne 2 7 . 0 1 . 2 0

Acknowledgements

First and foremost, I wish to express my sincere appreciation to my first supervisor, Prof. Dr. Hubert Randerath for giving me the opportunity to freely evolve as a scientist, whether it has been during my first research project or during this thesis. His fascination with graph theory has inspired me since my first attended lecture, and eventually resulted in the last two years of me studying this exciting research topic.

I also would like to thank my second supervisor, Dipl.-Math. Katharina Hammersen, who accepted the supervision of my proposed work without any hesitation.

A special thanks goes to Prof. Gunnar Brinkmann, who has given me many valuable hints for understanding the generation of simple quadrangulations of the sphere.

I wish to thank all the people of the Mathematics Stack Exchange and all the people of the Stack Overflow network for the free distribution of knowledge to the rest of the world. Many challenges in software engineering or mathematics could not be solved without your valuable help.

I would like to thank the inventor of coffee, whoever you are, this work would not have been possible without you.

I also wish to show my gratitude to my math teacher in school, Alice Schopp. A misunderstanding brought me into her course, and without her infectious passion for mathematics I would never have chosen the subject of computer science in the first place. We see each other in Banach space.

Last but not least, I would like to give a special thanks to my parents for supporting all my life choices, for putting the life of their children before their own for many years, and of course for always being on the phone.

Abstract

A graph is 1-planar if it can be embedded in the plane such that each edge cross at most once. If a 1-planar graph with n vertices has $4n-8$ edges, it is called *optimal* 1-planar. It is known, that optimal 1-planar graphs can be obtained by generating simple plane quadrangulations and then adding the crossing diagonals in each face. We implement an algorithm for generating optimal 1-planar graphs, measure its runtime and process the algorithms output to a conjecture-making program. Then, we automatically conjecture upper and lower bounds for the domination and the independence set number for optimal 1-planar graphs.

Zusammenfassung

Ein Graph heißt 1-planar, wenn er in die Ebene eingebettet werden kann, sodass sich jede seiner Kanten höchstens einmal kreuzt. Wenn ein 1-planarer Graph mit n Knoten $4n-8$ Kanten besitzt, so heißt er *optimal* 1-planar. Es ist bekannt, dass optimale 1-planare Graphen erzeugt werden können, indem in die Gebiete der einfachen Viereckgraphen die sich kreuzenden Diagonalen eingefügt werden. Wir implementieren einen Algorithmus für die Generierung der optimalen 1-planaren Graphen, messen dessen Laufzeit und leiten die generierten Graphen an ein Programm zur automatischen Erzeugung von Vermutungen weiter. Wir stellen obere und untere Schranken für die Dominanz- und Stabilitätszahl für diese Graphen auf.

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbrevations

CC	Crossed cube
cf.	confer
cp.	compare
CR	Crossed cube reduction
CS	Crossed star
e.g.	exempli gratia
et al.	et alii
etc.	et cetera
i.e.	id est
GI	Graph Isomorphism Problem
GCC	Gnu Compiler Collection
MIS	Maximum Independent Set
MSE	Mean Squared Error
NP-I	NP-Intermediate
O-P	Orientation-preserving
O-R	Orientation-reversing
RMSE	Root Mean Squared Error
SSE	Sum of Squares Due to Error
SR	Schumacher reduction
stdout	standard output

1 Introduction

> *The perception of an isomorphism between two known structures is a significant advance in knowledge- and I claim that it is such perceptions of isomorphism which create meanings in the minds of people.*
>
> —Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*

In addition to definitions and axiomatic systems, the process of discovering structural relationships or patterns in some kind of objects plays a fundamental role in mathematical research. A new conceivable relation is then formalized in a statement, which is initially open to formal proofs. These unproven statements are called *conjectures*; but we would not talk about mathematics, if there were no formal definition:

Definition 1.1. [47](*Conjecture*) An a priori hypothesis on the exactness or falseness of a statement of which one ignores the proof.

Even though conjectures have many shapes, they often describe a relation between different invariants of the object of interest, whereby an objects invariant is a property which "remains unchanged under certain classes of transformations" [85, 105]. Often, the conjectures are of the form $I \leq J, I \leq J + K$ and $I + J \leq K + L$ with I, J, K and L being either an invariant of the mathematical object or a constant value [28]. Due to the increasing complexity in modern mathematical research, the establishment of relations between these invariants in the above forms is becoming increasingly important. Therefore, in addition to pure scientific interest, researchers try to automate the process of conjecture-making using methods of computer science. Then, we also speak of *computerized mathematical discovery* [28]. As a matter of fact, the efforts of automated conjecture-making started as early as the 1980s with Fajtlowicz trying to understand what constitutes a good conjecture. He concluded in [38] that "a mathematical conjecture is more than a formula. Usually, it is also an expression of personal opinion concerning the significance, nontriviality, and correctness of this formula", which leads to one of the main difficulties of automated conjecture-making. We might ask: what makes some conjectures more significant than others? One can imagine that a program produces an endless stream of conjectures, having the form given above. Therefore, most conjecture-making programs "consist of various heuristics, whose purpose is deletion of trivial and otherwise noninteresting but true conjectures" [68]. Fajtlowicz presented an heuristic, called *Fajtlowicz-Dalmatian Heuristic*, tackeling this problem. It is implemented in *graffiti*, which is generally regarded as one of the first sources for automated conjecture-making. Since then, many programs have been written in order to automate the conjecture-making process, most famously a program called *conjecturing* by Larson and Van Cleemput.

Both authors are graph theorists and therefore mostly interested in graph theoretic conjecture-making. A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$. The elements of V are called vertices, the elements of E are called edges [32]. Two vertices, forming an edge, are called *adjacent*. Two graphs are called *isomorphic*, if there is an adjacency preserving bijection between their two vertex sets. Recalling the definition of an invariant, a *graph theoretic invariant* would be a property of a graph, which does not change under the transformation class of graph isomorphism. A simple example would be the number of vertices. Clearly, it does not change under isomorphism. There are many more graph theoretic invariants, e.g. the number of edges, the domination number or the independence number. A *dominating set* in a graph is a set D such that every vertex of the graph which is not in D is adjacent to at least one vertex in D. The *domination number* is the cardinality of a minimum dominating set [69]. A set of vertices is called an *independent set* if no two of its elements are adjacent [32]. The *independence number* is the cardinality of a minimum independent set. Calculating the independence or the domination number is proven to be $\mathcal{NP}$-complete, which is, despite its relevance for practical problems, precisely why they are so interesting for research. Since these invariants of interest are so hard to calculate, one is interested in upper and lower bounds, containing easy to calculate invariants. For example, the program *conjecturing* conjectured for a given simple graph $G = (V, E)$ the following upper bound for the domination number χ [69][1]

$$\chi \leq \frac{1}{2}|V| \tag{1}$$

In order to make conjectures, the program needs a set of graphs as an input, then calculate the invariants of interest, and finally make conjectures and check if they hold for each graph of the input set. Therefore, algorithms for generating graphs are essential. Generation algorithms are usually limited to a certain graph family, e.g. to planar graphs.

A graph is called *planar*, less formally speaking, if it can be embedded in the plane such that no of its edges cross. A *face* of a planar graph is an area enclosed by edges[2].

The research field of planar graph generation is well studied. Since each face of a planar graph is a simple polygon, studying generation algorithms began with the idea of generating polygonal structures. Berhard was the first scientist investigating this research topic in 1891. However, generating graph structures, according to today's comprehension, started much later due to the growing development of computer science. For example, Bowen and Fisk presented an algorithm in 1967 for generating triangulations, which are planar graphs, whereby each face is bounded by three edges. Barnette presented an algorithm for generating 4-connected and 5-connected planar graphs in 1974.

[1] This bound is well known and proven to be true [24].

[2] We are going to be much more precise in the next sections, explaining what we mean by the term *embedding* and *face*. For now, we leave it by this vague idea.

Brinkmann and McKay developed a program, called *plantri*, for the generation of certain subfamilies of planar graphs, e.g. planar triangulations, planar quadrangulations, planar cubic graphs and many more.

The process of generating graphs is closely related to the graph recognition problem, which asks whether a given graph can be assigned to a given graph family. In fact, if one finds an algorithm which generates all graphs of a given family, its reverse operations could solve the graph recognition problem for the given graph family. Recently, the graph recognition problem for a family of graphs, called 1-planar graphs, has attracted the interest of graph theoretic research. A 1-planar graph is a graph, for which each edge is crossed at most once. They were discovered by Ringel. It is proven that the graph recognition problem for 1-planar graphs in general is $\mathcal{NP}$-complete [44]. However, there are certain subfamilies of 1-planar graphs, for which efficient recognition algorithms exist. For example, Brandenburg presented an algorithm for recognizing optimal 1-planar graphs in linear time. A 1-planar graph is called *optimal*, if it has the maximum number of edges possible without destroying 1-planarity. They were introduced by Bodendiek et al. It is also shown that an optimal 1-planar graph with n vertices has exactly $4n - 8$ edges.
Furthermore, Auer et al. and Hong et al. developed an algorithm for solving the graph recognition problem for outer 1-planar graphs in linear time. A 1-planar graph is called *outer* 1-planar, if all its vertices are in the outer face [4].

For optimal 1-planar graphs Bodendiek et al. has shown an one-to-one correspondence between the class of optimal 1-planar graphs and the class of simple quadrangulations. This relationship is proven in the work of Fabrici and Madaras. The class of simple quadrangulations can be efficiently generated as shown by Brinkmann et al. Using his results and the relationship discovered by Bodendiek et al., Suzuki developed an algorithm for generating optimal 1-planar graphs. For the recognition problem, Brandenburg reverses the operations, used in the generation algorithm.

In the following thesis, we are going to develop an algorithm for generating optimal 1-planar graphs. We are going to use the generated graphs as an input for the program *conjecturing* in order to test the automated conjecture-making process for optimal 1-planar graphs. To do so, numerous topics need to be investigated.

We start by giving an introduction to graph theory in the second section, followed by an in-depth analysis of the graph isomorphism problem. Especially, we will see that graph isomorphism plays a vital role in both topics of graph generation and automated conjecture-making.

Then, we are going to introduce graph *planarity* using basic terms of general topology. We will give a precise definition of planar graphs and their embeddings, which are called *plane graphs*.

Additionally, we will give a brief introduction on how to represent a graph embedding using a combinatorial object called *rotation system*. Finally, we are going to introduce 1-*planarity*.

In the third section, we are going to give an in-depth investigation of optimal 1-planar graphs and their underlying structure, which is represented by the class of simple quadrangulations. We will see that optimal 1-planar graphs with n vertices only exist for $n = 8$ and $n \geq 10$, and that they do not exist for $n \leq 7$ and $n = 9$.

In the fourth section, we are going to introduce the topic of graph generation. Then, we are going to present the algorithm for generating simple quadrangulations as given by Brinkmann et al. Next, we are going to provide a guide on how to install and use the program *plantri*. Furthermore, we are going to setup *plantri* in the software environment *SageMath*, also called *sage*. We are going to do a performance comparison between the "plain" version of *plantri* and the one integrated in *sage*.

In the fifth section, we are going to develop and implement an algorithm for generating optimal 1-planar graphs. We are going to do a performance test of our implementation here as well.

In the sixth section, we are going to give a detailed introduction to the topic of automated conjecture-making. Especially, we are going to present the Fajtlowicz-Dalmatian heuristic for deciding whether a conjecture, made by the program, is significant or not. The heuristic is implemented in the program *conjecturing*, for which we will give an installation and usage guide. Furthermore, we will show how to process the generated graphs to *conjecturing* in order to make conjectures about various graph invariants of optimal 1-planar graphs.

Last but not least, we are going to summarize our results and give our final conclusions.

2 Theoretical principles

In the following sections we are going to give a brief overview of the definitions and theoretical concepts of graph theory. First, we are interested in graph isomorphism and topological graph theory. Both topics are mandatory for understanding the concept of graph generation.

Second, we will study the structure of planar and to a great extent 1-planar graphs. We are especially interested in optimal 1-planarity, and we are going to use the investigated structural properties of this graph family for developing a generation algorithm for optimal 1-planar graphs.

Furthermore, we require basic knowledge in algebra such as group theory, equivalence relations and linear mappings, which we will not explicitly define. The used definitions and ideas can be found in [15], [64],[39] and [57].

2.1 Graphs

Definition 2.1. [32, p.2](*Graph*) A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$. The elements of E are 2-element subsets of V. The elements of V are the *vertices* (or *nodes*) of the graph G, the elements of E are its *edges*.

A graph with vertex set V is said to be a graph on V. The vertex set of a graph G is referred to as $V(G)$, its edge set as $E(G)$. Instead of $G = (V, E)$ we often simply write G.

Definition 2.2. [32, p.2](*Order*) The number of vertices of a graph G is its *order*, written as $|G|$; its number of edges is denoted by $||G||$.

Definition 2.3. [32, p.2](*Finite graph*) A graph G is called *finite* if its order is finite. Otherwise, the graph is called *infinite*.

Definition 2.4. [32, p.2](*Trivial, empty graph*) For the *empty graph* $G = (\emptyset, \emptyset)$ we simply write $\emptyset$. A graph of order 0 or 1 is called *trivial graph*.

Definition 2.5. [32, p.3](*Adjacent, neighbor*) Two vertices v_i, v_j of a graph G are *adjacent*, or *neighbors*, if (v_i, v_j) is an edge of G.

Definition 2.6. [32, p.3](*Complete graph*) If all the vertices of a graph G are pairwise adjacent, then G is called *complete*. A complete graph on n vertices is a K_n.

Definition 2.7. [32, p.2](*Incident, ends*) A vertex v is *incident* with an edge e, if $v \in e$; then e is an edge at v. The two vertices incident with an edge are its *endvertices* or *ends*, and an edge *joins* its ends. We are going to abuse language to a certain extent, and sometimes say that an edge e is incident with a vertex v.

Definition 2.8. [14, p.3](*Loop, link*) An edge with identical ends is called a *loop*, and an edge with distinct ends a *link*.

Definition 2.9. [91, p.2](*Parallel edge*) Edges that have the same end vertices are called *parallel* edges.

Definition 2.10. [14, p.3](*Simple graph*) A graph is called *simple graph* if it has no loops and no parallel edges.

Definition 2.11. [14, p.5](*Bipartite graph*) A *bipartite graph* is one whose vertex set can be partitioned into two subsets X and Y, such that each edge has one end in X and one end in Y; such a partition (X, Y) is called a *bipartition* of the graph.

Definition 2.12. [32, p.5](*Degree*) The *degree* (or *valency*) $d_G(v) = d(v)$ of a vertex v is the number $|E(v)|$ of edges at v.

There is a special case for definition 2.12 concerning loops. A slightly different definition of the vertex degree is given by Bondy and Murty.

Definition 2.12a. [14, p.10](*Degree*) The degree $d_G(v) = d(v)$ of a vertex v is the number of edges incident with v, each loop counting as **two** edges.

The reasoning for why a loop counts as two edges is not really discussed in literature. However, by definition an edge is always incident with exactly two vertices, and hence a loop is incident to the same vertex two times. Furthermore, one can find some hints in the work of Diestel (cf. [32, p.5 and p.28]). One can reason that it follows from the *degree sum formula*, which is known in literature as the *handshaking lemma*.

Lemma 2.1. [14, p.10](*Handshaking lemma*) Given a simple graph $G = (V, E)$, whereby $|E|$ denotes the number of edges. Then, the following equation holds

$$\sum_{v \in V} d(v) = 2|E| \tag{2}$$

Proof. In a simple graph each edge consists of two vertices, which are by definition incident with that edge. The degree of a vertex is the number of edges for which that vertex is incident. Therefore each edge is counted twice if the vertex degrees are added up. □

In order to preserve correctness of the handshaking lemma for non-simple graphs, it is necessary to count each loop as two edges. We can also think of a loop as two ways of "leaving" a vertex, thus a vertex connected through a loop is adjacent to itself from both ends.

Example 2.13. In figure 2.1 two edges, namely $e_1 = (v_1, v_2)$ and $e_3 = (v_3, v_3)$, are shown. The arrows symbolize the incidence relation of an edge with its vertices, e.g. the arrow pointing from v_1 to e_1 reads as follows: v_1 is incident with e_1. Notice how v_3 can be interpreted to be adjacent to itself from both ends (clockwise and anti-clockwise). One can easily see that the handshaking lemma is only satisfied if $d(v_3) = 2$, therefore e_3 is counted twice.

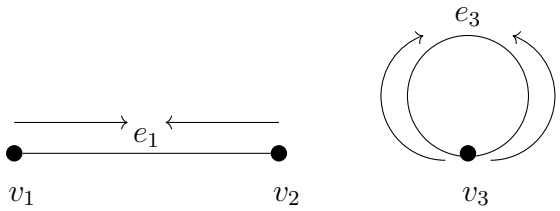


Figure 2.1: Example of the incidence relation.

Definition 2.14. [32, p.5](*Isolated vertex*) A vertex of degree 0 is called *isolated*.

Definition 2.15. [32, p.5](*Minimum degree*) The number $\delta(G) := min\{d(v) \mid v \in V\}$ of a graph $G = (V, E)$ is called the *minimum degree* of G.

Definition 2.16. [32, p.5](*Maximum degree*) The number $\Delta(G) := max\{d(v) \mid v \in V\}$ of a graph $G = (V, E)$ is called the *maximum degree* of G.

Definition 2.17. [32, p.5](*Average degree*) The number

$$d(G) := \frac{1}{|V|} \sum_{v \in V} d(v) \tag{3}$$

of a graph $G = (V, E)$ is called the *average degree* of G.

It is obvious that $\delta(G) \leq d(G) \leq \Delta(G)$.

Definition 2.18. [32, p.311](*Degree sequence*) Given a graph G with n vertices and degrees $d_1 \leq \cdots \leq d_n$. The n-tupel $(d_1, \cdots, d_n)$ is called the *degree sequence* of G.

Example 2.19. Given a graph $G = (V, E)$ with a set of vertices $V = \{v_1, \cdots, v_5\}$ and a set of edges $E = \{e_1, \cdots, e_5\}$ with $e_1 = (v_1, v_2), e_2 = (v_2, v_5), e_3 = (v_5, v_5), e_4 = (v_4, v_5)$, $e_5 = (v_4, v_5)$. The graph G is shown in figure 2.2.

Each vertex v is displayed as a dot and each edge e is pictured as a line between its two vertices. The edge $e_3 = (v_5, v_5)$ is a loop. The edges e_4 and e_5 are parallel since they share the same ends. The degree of each vertex is given by the number of its neighbors, e.g. the vertex v_2 has the degree $d(v_2) = 2$. Since the vertex v_3 has no neighbors, the degree is given by $d(v_3) = 0$, thus v_3 is isolated. Therefore, the minimum degree of G is $\delta(G) = 0$.

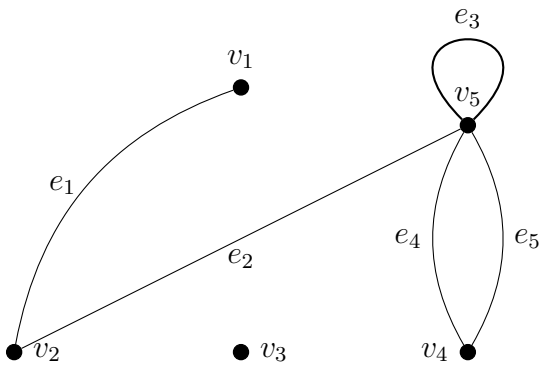


Figure 2.2: An example of a graph (cf. [91, p.2]).

The maximum degree of G is $\Delta(G) = 5$, given by $d(v_5) = 5$. v_5 is incident to e_2, e_3, e_4 and e_5. e_3 is a loop, so it is counted twice. The degree sequence is given by $(0, 1, 2, 2, 5)$.

Definition 2.20. [32, p.5](*Regular graph*) Given a graph $G = (V, E)$, whereby all vertices $v \in V$ have the same degree k, then G is *k-regular*, or simply regular. A 3-regular graph is called *cubic*

Example 2.21. Given a graph G with 10 vertices and 15 edges. Each vertex has exactly 3 neighbors, hence the degree sequence is given by $(3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$, and therefore, G is 3-regular (cubic). The graph is visualized in figure 2.3. The given graph G is also known as the "Petersen" graph [32, p.166].

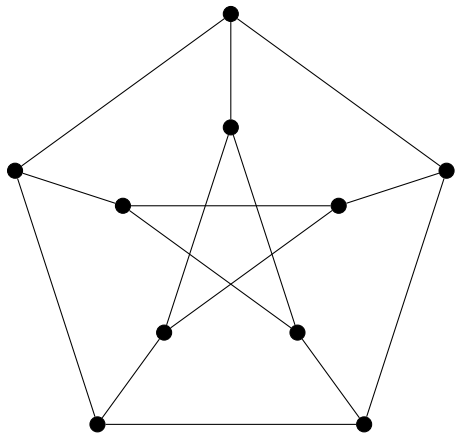


Figure 2.3: A 3-regular graph (Petersen graph).

Definition 2.22. [32, p.10](*Walk*) Given a graph G. A *walk* (of length k) in G is a non-empty alternating sequence $W = v_0e_0v_1e_1 \cdots e_{k-1}v_k$ of vertices and edges in G such that $e_i = (v_i, v_{i+1})\ \forall i < k$. If $v_0 = v_k$, the walk is *closed*. We say that W is a walk from v_0 to v_k [14, p.12], or a (v_0, v_k)-walk.

Definition 2.23. [14, p.12](*Trail*) Given a graph G with a walk $W = v_0e_0v_1e_1 \cdots e_{k-1}v_k$. If the edges $e_0, e_1, \cdots, e_{k-1}$ are distinct, W is called a *trail*.

Definition 2.24. [14, p.12](*Path*) Given a graph G with a trail $W = v_0e_0v_1e_1\cdots e_{k-1}v_k$. If the vertices $v_0, v_1, \cdots, v_k$ are distinct, W is called a *path*. We often denote a path by P instead of W.

Example 2.25. Given a graph $G = (V, E)$ with $V = \{v_0, v_1, v_2, v_3, v_4\}$ and $E = \{e_0 = (v_0, v_1), e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_0, v_3), e_4 = (v_1, v_3), e_5 = (v_3, v_4), e_6 = (v_2, v_4)\}$. The graph is shown in figure 2.4.

The sequence $W_1 = v_4e_6v_2e_2v_3e_2v_2$ is a walk. Notice that the edge e_2 is visited twice, therefore W_1 is not a trail. Since W_1 is not a trail, W_1 can not be a path. The sequence $W_2 = v_4e_6v_2e_2v_3e_4v_1e_1v_2$ is a trail, since all edges in the sequence are distinct. However, W_2 is not a path, since the vertex v_2 appears twice in the sequence. The sequence $W_3 = v_4e_6v_2e_1v_1e_0v_0e_3v_3$ is a path, since all edges and vertices in the sequence are distinct.

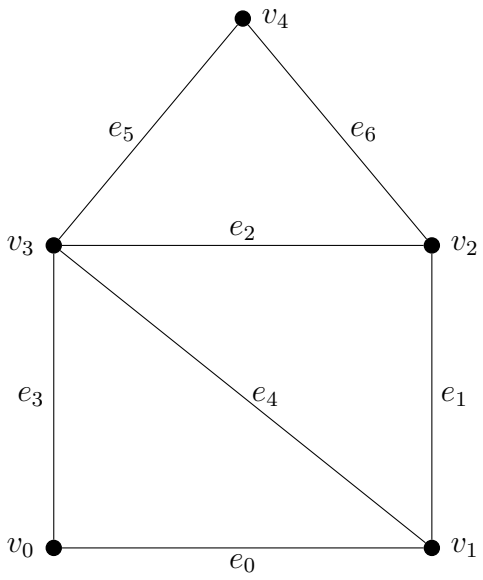


Figure 2.4: Example of a graph for illustrating a walk, a trail and a path (cf. [14, p.12]).

Definition 2.26. [32, p.8](*Cycle*) If $P = x_0 \cdots x_{k-1}$ is a path and $k \geq 3$, then the graph $C = P + x_{k-1}x_0$ is called a *cycle*. The length of a cylce is equal to the number of edges (or vertices); the cycle of length k is called a *k-cycle* and denoted by C_k.

Definition 2.27. [32, p.8](*Chord*) An edge which joins two vertices of a cycle but is not an edge of the cycle itself is a *chord* of that cycle.

Example 2.28. Given a graph G with 8 vertices and 8 edges. The graph is shown in figure 2.5. The edge $(x, y) \in E$ is a chord.

Definition 2.29. [32, p.135](*Chordal graph*) A graph is *chordal* (or triangulated) if each of its cycles with length of at least 4 has a chord, i.e. if it contains no induced cycles other than triangles.

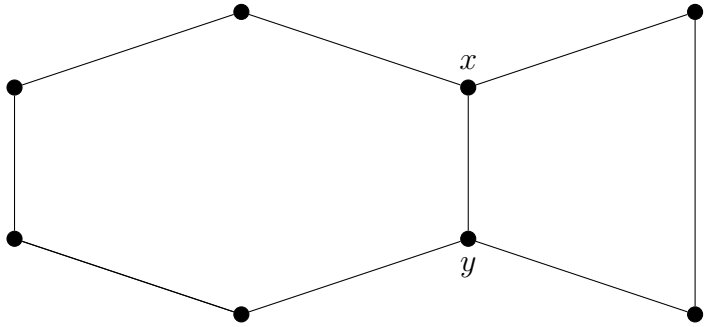


Figure 2.5: A cycle C_8 with chord (x, y) (cf. [32, p.8]).

Chordal graphs have, among others (e.g. *planar graphs*), a special property. The maximum independent set problem, as well as the maximum clique problem, which are proved to be $\mathcal{NP}$-complete [56, 61], is known to be polynomial-time solvable ([99], quoted after [42]).

Definition 2.30. [32, p.4](*Complement*) Given a graph $G = (V, E)$. The *complement* $\overline{G}$ of G is the graph on V with edge set $[V]^2 \setminus E$.

Example 2.31. Given two graphs G and G' with 5 vertices and 5 edges each. The two graphs are shown in figure 2.6. Notice how every edge of $\overline{G}$ is not an element of the edge set E of G. Generally speaking: the edge set E' of the complement graph $\overline{G}$ contains all possible edges of G, which are not part of the edge set E of G.

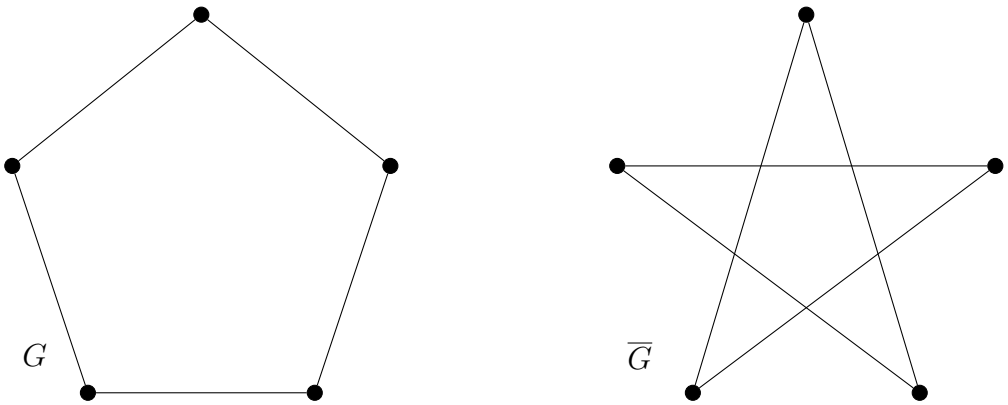


Figure 2.6: The complement $\overline{G}$ of G (cf. [32, p.4]).

Definition 2.32. [32, p.3-4](*Subgraph*) Given a graph G. We set $G \cup G' := (V \cup V', E \cup E')$ and $G \cap G' := (V \cap V', E \cap E')$. If $V' \subseteq V$ and $E' \subseteq E$, then G' is a *subgraph* of G, written as $G' \subseteq G$. Less formally, we say that G contains G'.

Example 2.33. Taking the graph drawn in figure 2.4. Now, let $G' = (V', E')$ with $V' = \{v_0, v_1, v_3\}$ and $E' = \{e_0, e_3, e_4\}$, then $V' \subseteq V = \{v_0, v_1, v_3\} \subset \{v_0, v_1, \cdots, v_4\}$ and $E' \subseteq E = \{e_0, e_3, e_4\} \subset \{e_0, e_1, \cdots, e_6\}$, thus $G' \subseteq G$.

Definition 2.34. [32, p.10](*Connected*) A non-empty graph G is called *connected* if any two of its vertices are linked by a path in G. Instead of "not connected" we usually say "disconnected".

Definition 2.35. [32, p.11](*Component*) Let $G = (V, E)$ be a graph. A maximal connected subgraph of G is a *component* of G. Since connected graphs are non-empty, the empty graph has no components.

Definition 2.36. [32, p.11-12](*k-connected*) A graph G is called *k-connected* for $k \in \mathbb{N}$ if $|G| > k$ and $G - X$ is connected for every set $X \subseteq V$ with $|X| < k$. In other words, no two vertices of G are separated by fewer than k other vertices. Every non-empty graph is 0-connected, and the 1-connected graphs are precisely the non-trivial connected graphs.

Definition 2.37. [32, p.12](*Connectivity*) The greatest integer k such that a graph G is k-connected is the *connectivity* $\kappa(G)$ of G.

Definition 2.38. [32, p.11](*Cutvertex*) A vertex which separates two other vertices of the same component is a *cutvertex*.

Definition 2.39. [32, p.11](*Bridge*) An edge which separates its ends is a *bridge*.

Example 2.40. In figure 2.7 a graph G with 3 components, named G_1, G_2 and G_3, is shown. G_1 is 1-connected, since the vertex v_x is a cutvertex. According to the definition, G_2 is 0-connected. G_3 is 2-connected, since we need to cut two vertices in order to separate G_3 into two components.

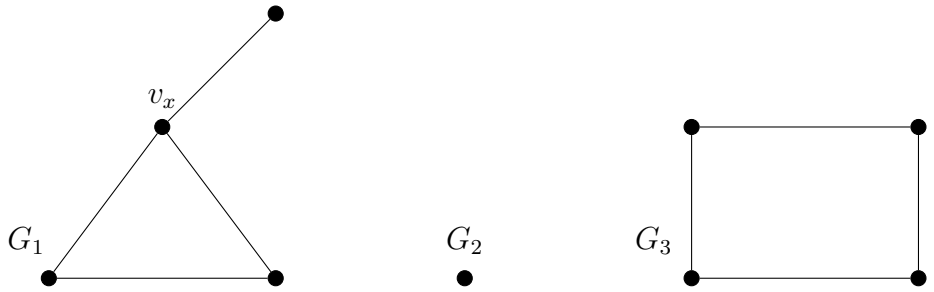


Figure 2.7: A graph G with three components G_1, G_2, G_3.

Definition 2.41. [95, p.8](*Clique, Clique Number*) Given a graph $G = (V, E)$, then a set $C \subseteq V$ of pairwise distinct adjacent vertices V of G is called *clique* of G, thus the subgraph of G induced by C is a complete graph. The *clique number* $\omega(G)$ denotes the size of a maximum clique in G.

Example 2.42. Given a graph $G = (V, E)$ with its set of vertices $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ and its set of edges $E = \{(v_0, v_1), (v_0, v_4), (v_1, v_4), (v_1, v_2), (v_3, v_4), (v_2, v_3), (v_3, v_5)\}$ as shown in figure 2.8.

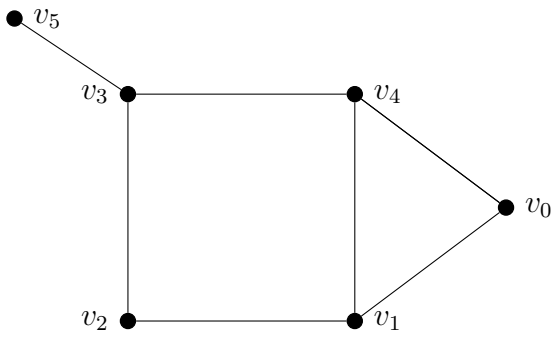


Figure 2.8: The maximum clique of the graph G, given by $\{v_0, v_1, v_4\}$ (cf. [95, p.8]).

The set $V' = \{v_0, v_1, v_4\} \subset V$ forms a clique in G. Notice how V' can not be extended by a vertex v_i of V, such that v_i is pairwise adjacent to all vertices of V', thus V' is a maximal clique of G.

Definition 2.43. [32, p.3,p.135](*Independent set, independence number*) A set of vertices of a graph G is called *independent* (or *stable*) if no two of its elements are adjacent. The stability, or *independence number* $\alpha(G)$ denotes the size of a maximum stable set in G.

An independent set D_i of a graph G is called maximal if and only if no vertex of G can be added to D_i without violating definition 2.43.
In the work of Lê, several approaches for finding the *Maximum Independent Set (MIS)* of a graph are discussed. In [56] it is proved that the MIS is $\mathcal{NP}$-complete.

The MIS is strongly connected to the maximum clique problem, which asks for the maximum clique in a graph. Notice: the complement of a *clique* forms an independent set.
In [71] it is stated: Given a graph $G = (V, E)$ and a vertex subset $S \subset V$. S is a maximum independent set if and only if S induces a maximum clique in $\overline{G}$. Hence, it is also considered as the dual version of the MIS problem.

It is obvious that the following relationship between the independence set number and the clique number holds: $\alpha(G) = \omega(\overline{G})$ and $\omega(G) = \alpha(\overline{G})$ (cf. [32, p.135]).

Definition 2.44. [32, p. 27](*Adjacency matrix*) The *adjacency matrix* $A = (a_{ij})_{n\times n}$ of a graph $G = (V, E)$ with n vertices is defined by

$$a_{ij} := \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Definition 2.45. [32, p. 27](*Incidence matrix*) The *incidence matrix* $B = (b_{ij})_{n\times m}$ of a graph $G = (V, E)$ with $V = \{v_1, \cdots, v_n\}$ and $E = \{e_1, \cdots, e_m\}$ is defined by

$$b_{ij} := \begin{cases} 1 & \text{if } v_i \in e_j \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

2.2 Graph isomorphism

In the following section we are going to investigate a subject in graph theory called *graph isomorphism*, which deals with the problem of when two graphs are considered identical. It is of great interest for the generation of graphs and we will heavily use the results of this section in our generation algorithms.

Definition 2.46. [46, p.4](*Graph homomorphism*) Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. A vertex function $\varphi : V \to V'$ *preserves adjacency* if

$$(x, y) \in E \iff (\varphi(x), \varphi(y)) \in E' \ \forall x, y \in V \tag{6}$$

Similarly, φ *preserves non-adjacency* if

$$(x, y) \notin E \iff (\varphi(x), \varphi(y)) \notin E' \ \forall x, y \in V \tag{7}$$

φ is called an *homomorphism* from G to G'. We say that G is homomorphic to G'.

Definition 2.47. [46, p.4](*Structure preserving*) Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. A vertex bijection $\varphi : V \to V'$ is *structure preserving*, if it preserves adjacency and non-adjacency.

Definition 2.47 lead to the definition of graph isomorphism.

Definition 2.48. [46, p.5](*Graph isomorphism*) Two graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic*, denoted $G \simeq G'$, if $\exists$ a structure-preserving bijection $\varphi : V \to V'$.
Such a bijection φ is called an *isomorphism* from G to G'.

In the work of Diestel a more compact definition is given.

Definition 2.48a. (Cf. [32, p.3])(*Graph isomorphism*) Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We call G and G' *isomorphic*, and write $G \simeq G'$, if there exists a bijection $\varphi : V \to V'$ with $(x, y) \in E \iff (\varphi(x), \varphi(y)) \in E' \ \forall x, y \in V$. Such a map φ is called an *isomorphism*.

Notice that the definitions 2.48 and 2.48a are not sufficient if non-simple graphs are considered. However, we can fix the definition in order to preserve correctness for non-simple graphs.

Definition 2.48b. [46, p.9](*Structure preserving*) Given two graphs $G = (V, E)$ and $G' = (V', E')$. A vertex bijection $\varphi : V \to V'$ is *structure preserving* if

(i) φ is a graph homomorphism.

(ii) The number of edges between $v_x, v_y \in G$ is equal to the number of edges between $\varphi(v_x), \varphi(v_y) \in G', \forall v_x, v_y \in V, v_x \neq v_y$.

(iii) The number of loops at each vertex $v \in G$ is equal to the number of loops at the vertex $\varphi(v) \in G', \forall v \in V$.

In the following we are going to abuse language to a certain extent. Sometimes, we are going to call graph isomorphism and graph homomorphism just isomorphism and homomorphism, respectively.

Example 2.49. Given two graphs $G = (V, E)$ and $G' = (V', E')$ with $V = (v_0, v_1, v_2), E = \{(v_0, v_1), (v_0, v_2), (v_1, v_2)\}$ and $V' = \{v'_0, v'_1, v'_2\}, E' = \{(v'_0, v'_1), (v'_0, v'_2), (v'_1, v'_2), (v'_1, v'_2)\}$. The two graphs are shown in figure 2.9.

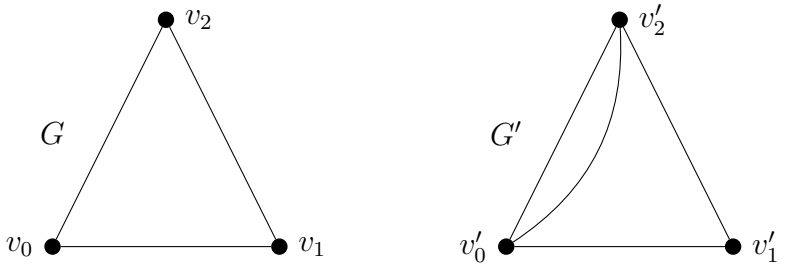


Figure 2.9: Error in definition 2.48a of graph isomorphism when considering non-simple graphs (cf. [46, p.10]).

According to definition 2.48a G and G' would be isomorphic, but they clearly have not the same structure. Definition 2.48b fixes the issue.

In the following we assume, unless explicitly stated, that all given graphs are simple.

Less formally, the idea of graph isomorphism is to tell whether two given graphs G and G' are identical. The computational problem to test whether two graphs are isomorphic or not is called the *Graph Isomorphism Problem (GI)* [88, p.1].

The GI is one of the most researched problems in theoretical computer science with great theoretical interest, since its complexity still remains unkown (cf. [92, p.12]). In fact, the problem is even discussed as a candidate for modern quantum computing [41, 81]. It is believed that the GI is neither $\mathcal{NP}$-complete nor in $\mathcal{P}$, which would make it a candidate for a $\mathcal{NP}$-intermediate ($\mathcal{NP}$-I), according to Ladner's theorem under the assumption that $\mathcal{NP} \neq \mathcal{P}$ (cf. [66],[92, p.64]).

In the work of Babai a quasipolynomial time algorithm for general graphs and in the work of Hopcroft and Wong a linear time algorithm for planar graphs for the GI is presented (cf. [101]).

Apart from a theoretical point of view, the GI has many practical applications as well, such as image processing or matching protein structures in molecular graphs [93].

There are certain properties of a graph which must be preserved under isomorphism and which are often easy to calculate. These properties are called *graph invariants*.

Definition 2.50. [6, p.15](*Graph invariant*) Given two graphs G and G'. Let G and G' be isomorphic. A *graph invariant* is a function f such that $f(G) = f(G')$.

Notice that $f(G) = f(G')$ does not mean that G and G' are isomorphic, but it is a necessary condition. If an invariant is both necessary and sufficient, it is said to be *complete*. We also call a complete graph invariant a *certificate* [6, p.15]. If two graphs G and G' share a common complete graph invariant, they are isomorphic.

Given two graphs $G = (V, E)$ and $G' = (V', E')$. Let $\varphi : V \to V'$ be an isomorphism from G to G'. The following statements, without the claim of completeness, are isomorphic graph invariants of G.

Corollary 2.1. The number of vertices of G.

Proof. Since an isomorphism from G to G' is a bijection and a bijection is by definition injective and surjective[3], $|V| = |V'|$ must hold. □

Corollary 2.2. The number of edges of G.

Proof. Since an isomorphism is a bijection between the vertex sets of G and G' and that bijection is structure preserving, $|E| = |E'|$ must hold. □

Corollary 2.3. The number of vertices of a given degree k.

Proof. We define a function

$$f_{deg} : v \mapsto \begin{cases} 1 & \text{if } deg(v) = k \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

By definition a vertex $\varphi(v) \in V'$ is incident with an edge $e' \in E'$ if and only if $v \in V$ is incident with an edge $e \in E$, thus $\sum_{v \in V} f_{deg}(v) = \sum_{v' \in V'} f_{deg}(v')$. □

[3]Cf. definition A.2, A.3 and A.4 in the appendix.

Corollary 2.4. G is bipartite.

Proof. Given two graphs $G = (V, E)$ and $G' = (V', E')$ and an isomorphism $\varphi : V \to V'$ from G to G'. Let G be bipartite. Then, by definition, $V = V_1 \cup V_2$ with $V_1 \cap V_2 = \emptyset$ such that for each $e = (v_x, v_y) \in E : v_x$ is in V_1 and v_y is in V_2. φ is a bijection, hence there must be two vertex sets V_1' and V_2' such that there exists $V_1 \to V_1'$ and $V_2 \to V_2'$ with $V' = V_1' \cup V_2'$ and $V_1' \cap V_2' = \emptyset$. For any edge $e = (v_x, v_y)$, there must be an edge $e' = (\varphi(v_x), \varphi(v_y))$ with $\varphi(v_x) \in V_1'$ and $\varphi(v_y) \in V_2'$. Therefore, G' is bipartite. □

Example 2.51. Given two graphs $G = (V, E)$ with $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$ and $E = \{(v_0, v_1), (v_0, v_9), (v_0, v_7), (v_1, v_2), (v_1, v_8), (v_2, v_9), (v_2, v_3), (v_3, v_4), (v_3, v_8), (v_4, v_5), (v_4, v_9), (v_5, v_8), (v_5, v_6), (v_6, v_7), (v_6, v_9), (v_7, v_8)\}$ and $G' = (V', E')$ with $V' = \{v_0', v_1', v_2', v_3', v_4', v_5', v_6', v_7', v_8', v_9'\}$ and $E' = \{(v_0', v_1'), (v_0', v_9'), (v_0', v_7'), (v_1', v_2'), (v_1', v_8'), (v_2', v_9'), (v_2', v_3'), (v_3', v_4'), (v_3', v_8'), (v_4', v_5'), (v_4', v_9'), (v_5', v_8'), (v_5', v_6'), (v_6', v_7'), (v_6', v_9'), (v_7', v_8')\}$.
One possible embedding of the two graphs is shown in figure 2.10.

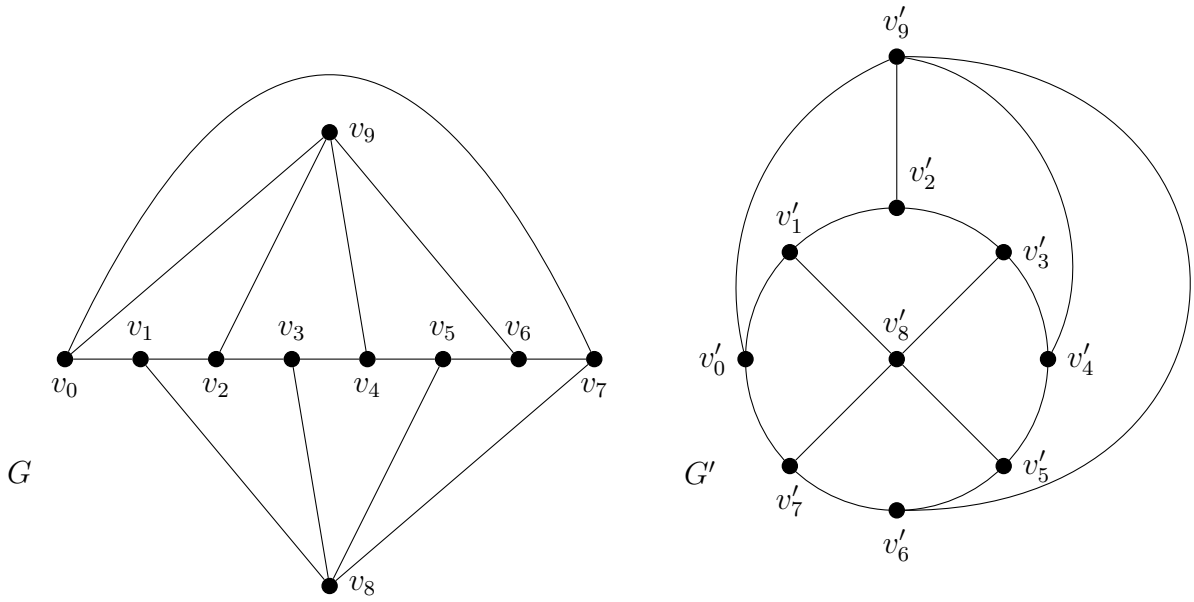


Figure 2.10: Two graphs G and G' being isomorphic.

Now, we want to check if G and G' are isomorphic. To do so, we need to find a structure preserving vertex bijection φ. The vertex function from V to V' is given by $\varphi : V \to V'$ with $\varphi(v_0) = v_0', \varphi(v_1) = v_1', \varphi(v_2) = v_2', \varphi(v_3) = v_3', \varphi(v_4) = v_4', \varphi(v_5) = v_5', \varphi(v_6) = v_6', \varphi(v_7) = v_7', \varphi(v_8) = v_8', \varphi(v_9) = v_9'$. The vertex function φ is clearly bijective.

The adjacency matrix A of G is given by

$$A = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \end{array} \begin{array}{c} \begin{array}{cccccccccc} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{9}$$

The adjacency matrix A' of G' is given by

$$A' = \begin{array}{c} \\ v'_0 \\ v'_1 \\ v'_2 \\ v'_3 \\ v'_4 \\ v'_5 \\ v'_6 \\ v'_7 \\ v'_8 \\ v'_9 \end{array} \begin{array}{c} \begin{array}{cccccccccc} v'_0 & v'_1 & v'_2 & v'_3 & v'_4 & v'_5 & v'_6 & v'_7 & v'_8 & v'_9 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{10}$$

It holds that $A = A'$, thus G and G' are isomorphic. In order to give a more intuitive solution for the problem, we are going to graphically convert G into G'. First, we are going to introduce two axes named x and y. The two axes are orthogonal to each other and they intersect in v_8. The two axes divide the drawing into four regions (compare with figure 2.11).

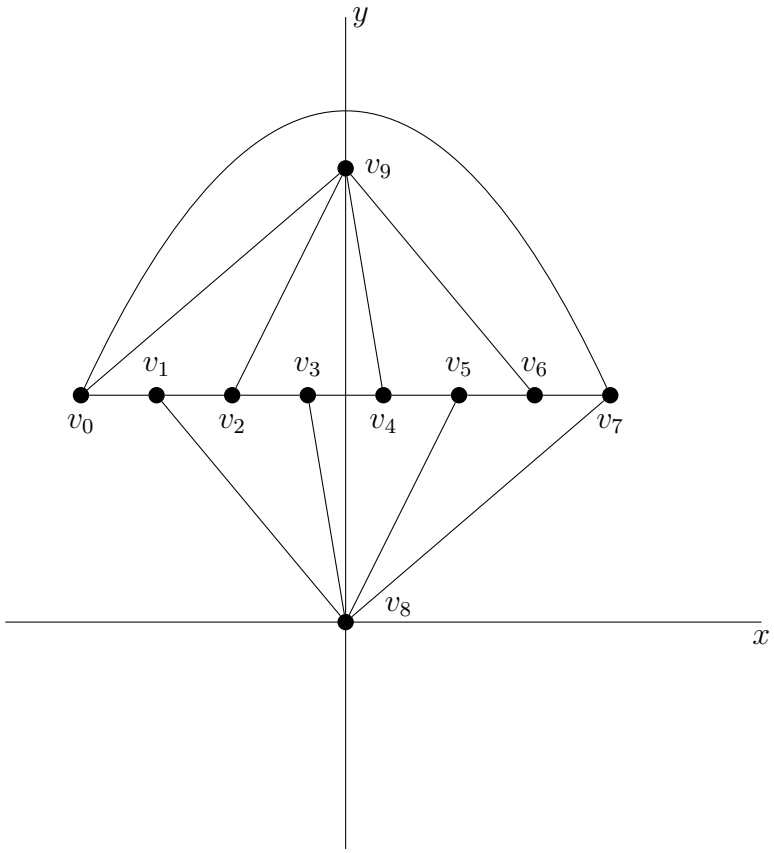


Figure 2.11: Adding two orthogonal axes to the embedding of G.

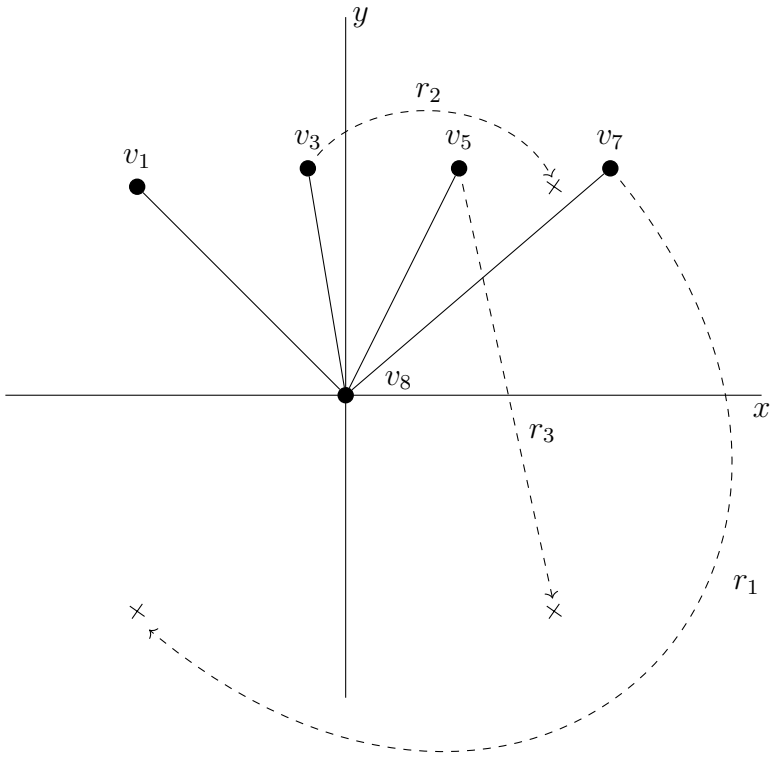
Second, we omit all vertices except for v_1, v_3, v_5, v_7, v_8. Furthermore, we take care of symmetry. Next, we perform the following movements (compare with figure 2.12a):

1. Move v_7 to the reflection point of v_1 with respect to the x-axis. We call that movement r_1.

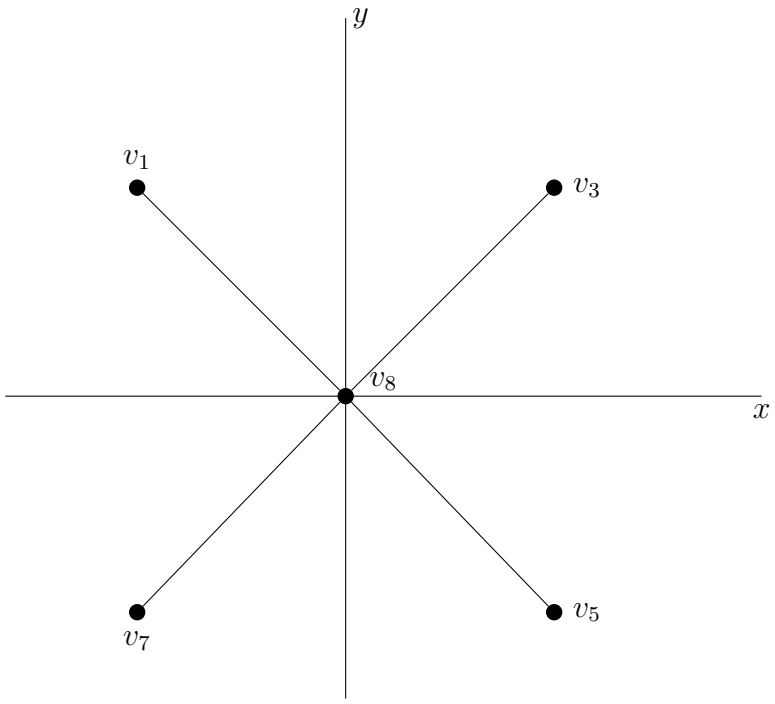
2. Move v_3 to the reflection point of v_1 with respect to the y-axis. We call that movement r_2.

3. Move v_5 to the reflection point of v_3 (after step 2) with respect to the x-axis. We call that movement r_3.

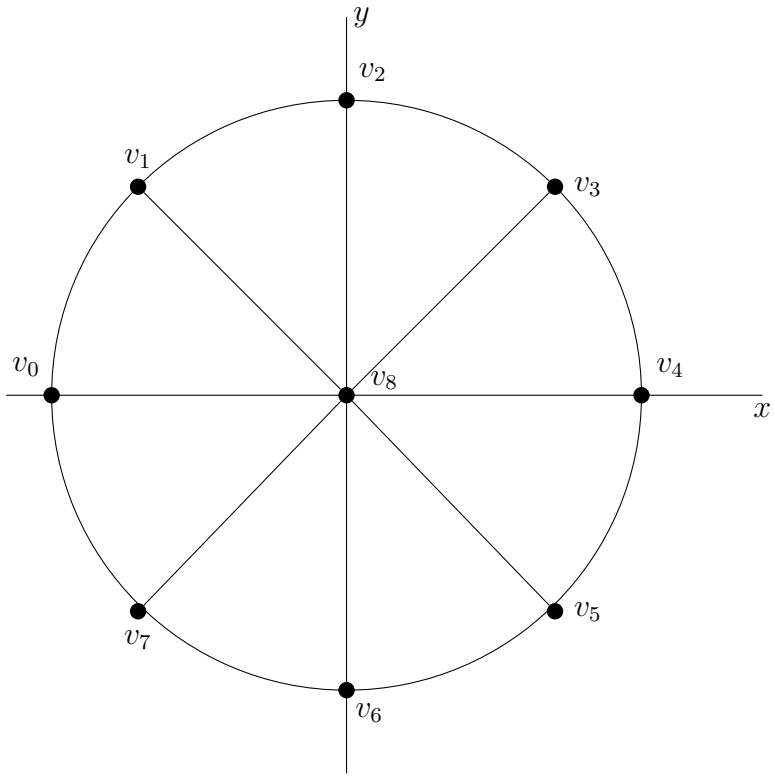
The resulting graph is shown in figure 2.12b. Next, we add the vertex v_0 in between v_1 and v_7, the vertex v_2 in between v_1 and v_3, the vertex v_4 in between v_3 and v_5, as well as the vertex v_6 in between v_7 and v_5. All vertices lay on a circle with radius equal to the length of the arc between v_8 and v_1. We add all the omitted edges as well (compare with figure 2.12c).
In the next step, we remove the two axes x and y from the first step, and add the vertex v_9 above vertex v_2. We add all the omitted edges as well (compare with figure 2.12d). In the last step, we rename the vertices according to the vertex bijection φ, which we have found previously (compare with figure 2.12e).



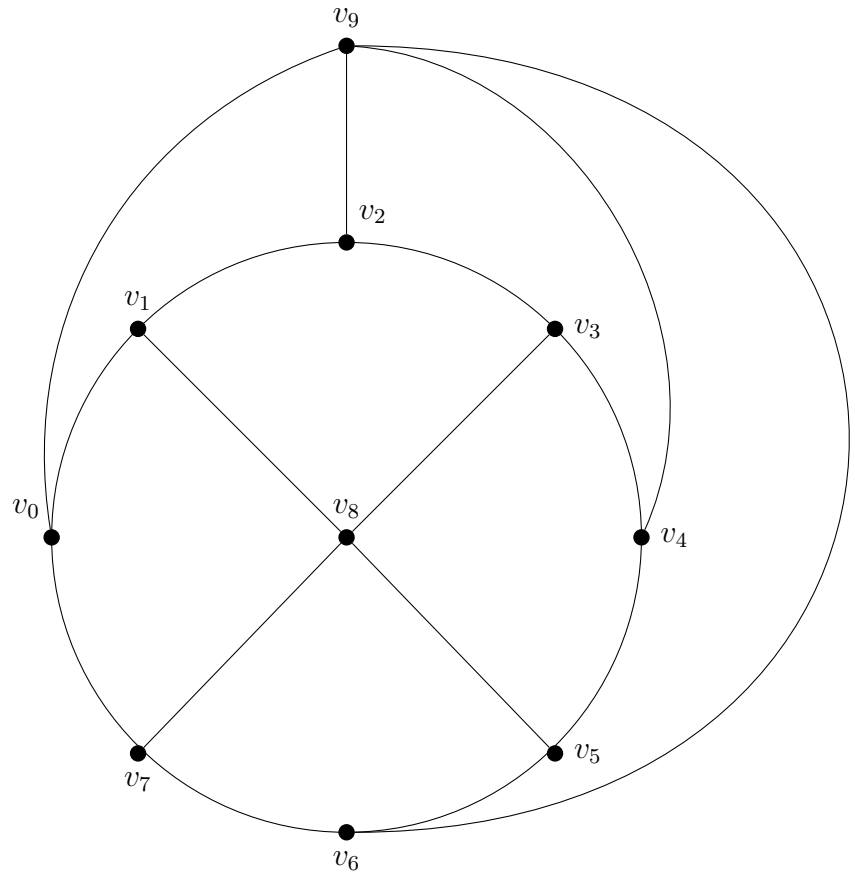
(a) Omitting vertices and perform reflections.



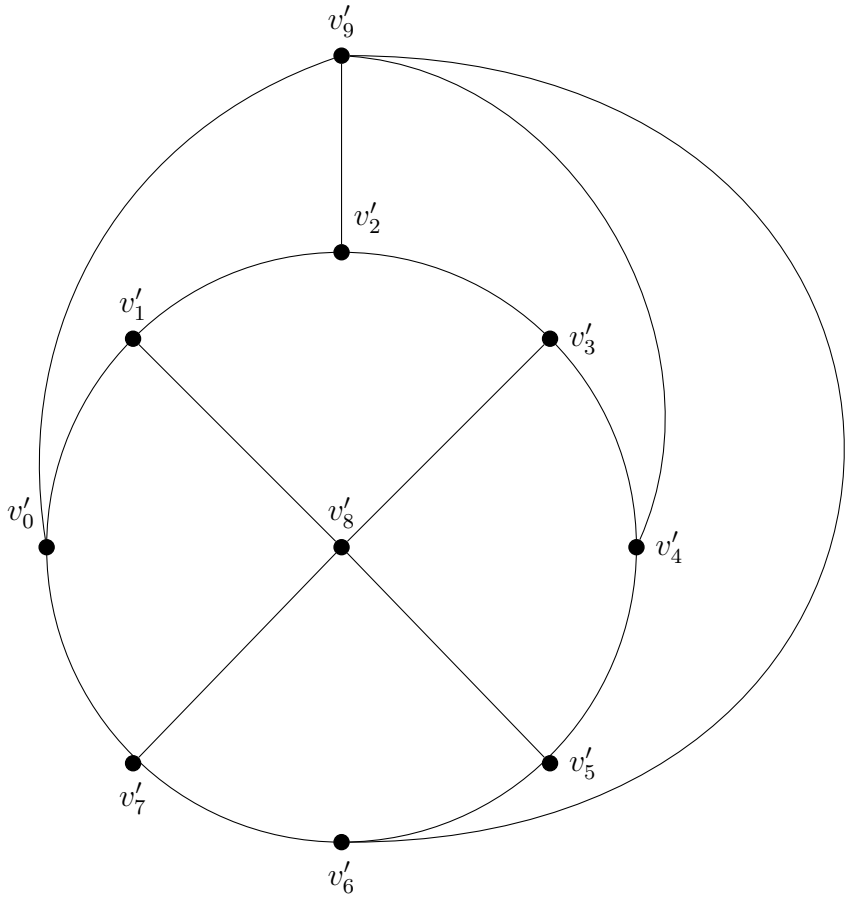
(b) Resulting graph of the first step.



(c) Adding omitted vertices and edges in a circle around v_8.



(d) Removing the axes and add vertex v_9 with its edges.



(e) Renaming vertices.

Figure 2.12: Proving graph isomorphism from G to G'.

In the following we are going to explore more important characteristics of graph isomorphism, which are of great interest for our generation algorithm.

Theorem 2.1. Graph isomorphism is an equivalence relation.[4]

Proof. According to the definition, we need to prove reflexivity, symmetry and transitivity.

(i) *Reflexivity*: Given a graph $G = (V, E)$. We define the vertex function as

$$\varphi : V \to V : \forall v \in V : \varphi(v) = v \tag{11}$$

We need to show that φ is a bijection. It holds $\forall v_x, v_y \in V : \varphi(v_x) = \varphi(v_y) \Rightarrow v_x = v_y$ since by definition $\varphi(v_x) = v_x$ and $\varphi(v_y) = v_y$, hence φ is injective.
$\forall v \in V, \exists v \in V : \varphi(v) = v$, which is true by definition, hence φ is surjective. Therefore, φ is bijective. φ is also structure preserving, since

$$(v_x, v_y) \in E \Leftrightarrow (\varphi(v_x), \varphi(v_y)) = (v_x, v_y) \in E, \quad \forall v_x, v_y \in V \tag{12}$$

[4]See A.7 in the appendix for the definition.

Therefore, $G \simeq G$, making $\simeq$ reflexive.

(ii) *Symmetry*: Given two graphs $G = (V, E)$ and $G' = (V', E')$. Let $G \simeq G'$, then there is a bijection $\varphi : V \to V'$. Then $\exists\ \varphi^{-1} : V' \to V$, which is bijective[5]. We need to show that

$$(v'_x, v'_y) \in E' \Leftrightarrow (\varphi^{-1}(v'_x), \varphi^{-1}(v'_y)) \in E,\ \forall v'_x, v'_y \in V' \tag{13}$$

holds. Since φ is bijective, there exists a mapping such that $\forall v'_x, v'_y \in V', \exists v_x, v_y \in V : \varphi(v_x) = v'_x, \varphi(v_y) = v'_y$, hence

$$(\varphi(v_x), \varphi(v_y)) \in E' \Leftrightarrow (v_x, v_y) \in E, \qquad \forall v_x, v_y \in V \tag{14}$$

$$\Leftrightarrow (\varphi(v_x), \varphi(v_y)) \in E' \Leftrightarrow (\varphi^{-1}(\varphi(v_x)), \varphi^{-1}(\varphi(v_y))) \in E, \forall v_x, v_y \in V \tag{15}$$

$$\Leftrightarrow \qquad (v'_x, v'_y) \in E' \Leftrightarrow (\varphi^{-1}(v'_x), \varphi^{-1}(v'_y)) \in E, \qquad \forall v'_x, v'_y \in V' \tag{16}$$

Therefore, φ^{-1} is structure preserving, and thus $G \simeq G' \Rightarrow G' \simeq G$.

(iii) *Transitivity*: Given three graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ and $G_3 = (V_3, E_3)$. Let $G_1 \simeq G_2$ and $G_2 \simeq G_3$. Then, there are two structure preserving bijections $\varphi_{12} : V_1 \to V_2$ and $\varphi_{23} : V_2 \to V_3$. Define $\varphi_{13} : V_1 \to V_3 : \varphi_{23} \circ \varphi_{12}$. Since φ_{12} and φ_{23} are bijections, their composition $\varphi_{13} : \varphi_{23} \circ \varphi_{12}$ is also a bijection[6]. We need to show that $\varphi_{23} \circ \varphi_{12}$ is structure preserving. It holds that

$$(v_{x1}, v_{y1}) \in E_1 \Leftrightarrow (\varphi_{12}(v_{x1}), \varphi_{12}(v_{y1})) \in E_2, \forall v_{x1}, v_{y1} \in V_1 \tag{17}$$

$$(v_{x2}, v_{y2}) \in E_2 \Leftrightarrow (\varphi_{23}(v_{x2}), \varphi_{23}(v_{y2})) \in E_3, \forall v_{x2}, v_{y2} \in V_2 \tag{18}$$

Since φ_{12} is bijective, there exists a mapping such that $\forall v_{x2}, v_{y2} \in V_2, \exists v_{x1}, v_{y1} \in V_1 : \varphi_{12}(v_{x1}) = v_{x2}, \varphi_{12}(v_{y1}) = v_{y2}$, hence

$$(\varphi_{12}(v_{x1}), \varphi_{12}(v_{y1})) \in E_2 \Leftrightarrow (\varphi_{23}(\varphi_{12}(v_{x1})), \varphi_{23}(\varphi_{12}(v_{y1}))) \in E_3, \forall v_{x1}, v_{y1} \in V_1 \tag{19}$$

$$\Rightarrow \qquad (v_{x1}, v_{y1}) \in E_1 \Leftrightarrow (\varphi_{23}(\varphi_{12}(v_{x1})), \varphi_{23}(\varphi_{12}(v_{y1}))) \in E_3, \forall v_{x1}, v_{y1} \in V_1 \tag{20}$$

Therefore, $\varphi_{13} : V_1 \to V_3 : \varphi_{23} \circ \varphi_{12}$ is structure preserving, and thus

$$G_1 \simeq G_2 \wedge G_2 \simeq G_3 \Rightarrow G_1 \simeq G_3. \tag{21}$$

From $(i), (ii)$ and (iii) follows that $\simeq$, which denotes graph isomorphism, is an *equivalence relation.* □

[5]See theorem A.1 in the appendix.
[6]See theorem A.2 in the appendix.

Since graph isomorphism is an equivalence relation, we can divide a given set $\mathcal{G}$ of graphs into subsets, called equivalence classes.

Definition 2.52. (*Graph isomorphism class*) Given an isomorphism relation $\simeq$ on a set $\mathcal{G}$ of graphs and a graph $G \in \mathcal{G}$, we define a certain subset $\mathcal{C}_G$ of $\mathcal{G}$ by

$$\mathcal{C}_G = \{G' \mid G' \in \mathcal{G} \text{ and } G' \simeq G\} \tag{22}$$

We call $\mathcal{C}_G$ the *isomorphism class* determined by G.[7]

Lemma 2.2. Given a non-empty set $\mathcal{G}$ of graphs. $G \simeq G' \Leftrightarrow \mathcal{C}_G = \mathcal{C}_{G'}$ with $G, G' \in \mathcal{G}$.

Proof. We prove " $\Rightarrow$ " and " $\Leftarrow$ ".[8]

(i) Assume $G \simeq G'$.

$$G_x \in \mathcal{C}_G \Rightarrow G_x \simeq G \Rightarrow G_x \simeq G' \text{ (transitivity)} \tag{23}$$
$$\Rightarrow G_x \in \mathcal{C}_{G'},\ \forall G_x \in \mathcal{C}_G \tag{24}$$

Therefore, $\mathcal{C}_G \subseteq \mathcal{C}_{G'}$; analogously, $\mathcal{C}_{G'} \subseteq \mathcal{C}_G$, hence $\mathcal{C}_G = \mathcal{C}_{G'}$.

(ii) Assume $\mathcal{C}_G = \mathcal{C}_{G'}$.

$$G \simeq G \Rightarrow G \in \mathcal{C}_G \text{ (reflexivity)} \tag{25}$$
$$\Rightarrow G \in \mathcal{C}_{G'} \Rightarrow G \simeq G' \tag{26}$$

□

Now, we can assign each graph to exactly one equivalence class in terms of isomorphism. This natural classification of graphs is going to be useful in the implementation of our generation algorithms as we want to pick one graph for each isomorphism class and reject all the others. For that purpose, we are going use an algorithm of the program *nauty* as described in the work of McKay and Piperno.

A special case of isomorphism is called *automorphism*, which is an isomorphism from a graph to itself.

Definition 2.53. [32, p.3](*Graph automorphism*) Given two graphs G and G'. Let $G \simeq G'$, then there is an isomorphism φ from G to G'. If $G = G'$, φ is called *graph automorphism*.

[7] Adapted from definition A.8.
[8] Proof adopted from [86] and adapted so that it is consistent with our definitions.

In other words: given a graph $G = (V, E)$, then an automorphism from G to itself is defined as an isomorphism $\varphi : V \to V$. Obviously φ must be structure preserving by definition.

Example 2.54. Given a graph $G = (V, E)$ with $V = \{v_0, v_1, v_2\}$ and $E = \{(v_0, v_1), (v_0, v_2), (v_1, v_2)\}$. Let $G \simeq G$ with $\varphi : V \to V$. The isomorphism φ is an automorphism from G to G. In this example the function φ is the identity function. The graphs are shown in figure 2.13.

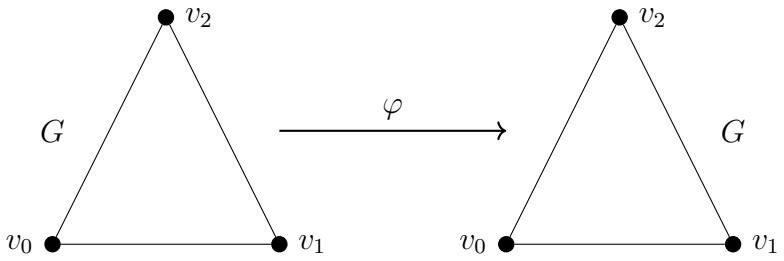


Figure 2.13: Example of a graph automorphism.

Taking the graph G from example 2.54, we can find other graphs which are automorph to G.

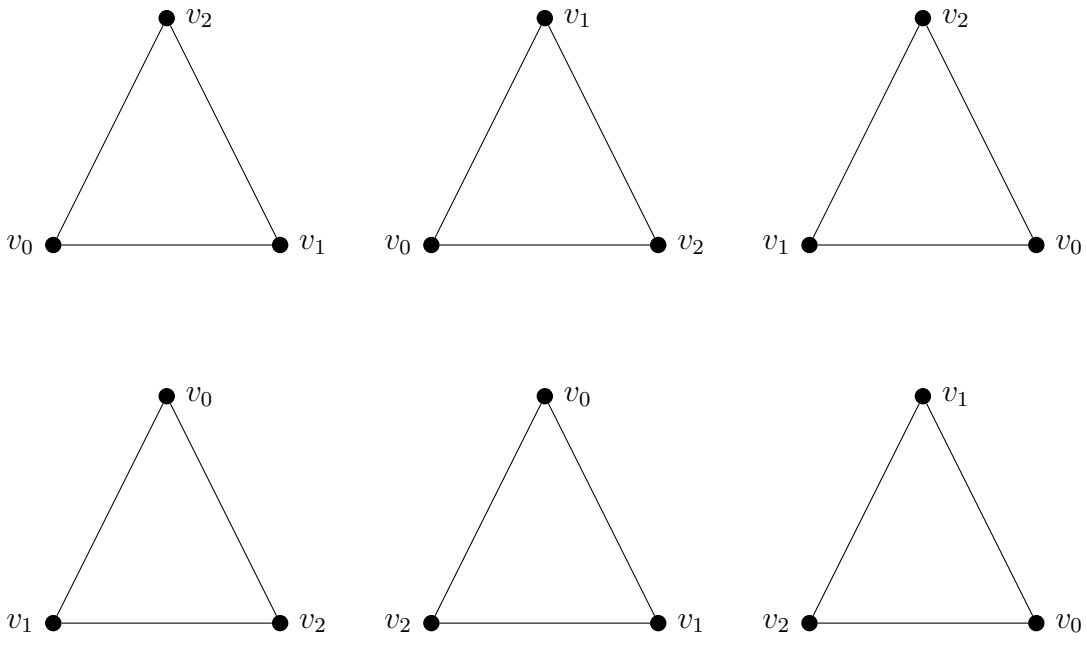


Figure 2.14: All graphs being automorph to G.

In figure 2.14 all graphs which are automorph to G are shown. If we take a closer look, we notice that the automorphisms of G are the permutations of its vertices.

Definition 2.55. [90, p.2](Graph automorphism) A *graph automorphism* of a graph $G = (V, E)$ is a permutation ϕ on the set of vertices V that satisfies the property that

$$(v_i, v_j) \in E \Leftrightarrow (\phi(v_i), \phi(v_j)) \in E, \ \forall v_i, v_j \in V \tag{27}$$

We denote the set of all automorphisms of G by $Aut(G)$.

Example 2.56. Taking the graph $G = (V, E)$ from example 2.54. The permutations, which generate all automorphisms of G, are given by

$$\phi_0 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_0 & v_1 & v_2 \end{pmatrix} \phi_1 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_0 & v_2 & v_1 \end{pmatrix} \phi_2 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_1 & v_0 & v_2 \end{pmatrix} \phi_3 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_1 & v_2 & v_0 \end{pmatrix} \tag{28}$$

$$\phi_4 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_2 & v_1 & v_0 \end{pmatrix} \phi_5 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_2 & v_0 & v_1 \end{pmatrix} \tag{29}$$

The set of all automorphisms of G is then given by the permutations given above, thus $Aut(G)$ is defined by $\phi_0, \phi_1, \phi_2, \phi_3, \phi_4$ and ϕ_5.

If we compose two of the permutations, we get a permutation which is again a graph automorphism of G.

Example 2.57. In the following, we are going to compose some of the permutations given above.
Composing ϕ_1 with ϕ_2 yields in ϕ_5.

$$\phi_1 \circ \phi_2 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_0 & v_2 & v_1 \end{pmatrix} \circ \begin{pmatrix} v_0 & v_1 & v_2 \\ v_1 & v_0 & v_2 \end{pmatrix} = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_2 & v_0 & v_1 \end{pmatrix} = \phi_5 \tag{30}$$

Composing ϕ_2 with ϕ_3 yields in ϕ_1.

$$\phi_2 \circ \phi_3 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_1 & v_0 & v_2 \end{pmatrix} \circ \begin{pmatrix} v_0 & v_1 & v_2 \\ v_1 & v_2 & v_0 \end{pmatrix} = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_0 & v_2 & v_1 \end{pmatrix} = \phi_1 \tag{31}$$

Composing a permutation with itself yields obviously in the permutation itself.

$$\phi_1 \circ \phi_1 = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_0 & v_2 & v_1 \end{pmatrix} \circ \begin{pmatrix} v_0 & v_1 & v_2 \\ v_0 & v_2 & v_1 \end{pmatrix} = \begin{pmatrix} v_0 & v_1 & v_2 \\ v_0 & v_2 & v_1 \end{pmatrix} = \phi_1 \tag{32}$$

Definition 2.58. (Cf. [64, p.15])(*Identity permutation*) A permutation ϕ of n vertices $\{v_0, v_1, \cdots, v_n\}$ of a graph G given by

$$id_G = \begin{pmatrix} v_0 & v_1 & \cdots & v_n \\ v_0 & v_1 & \cdots & v_n \end{pmatrix} \tag{33}$$

is called the *identity permutation*.

The identity permutation is obviously an automorphism of a graph G, since it describes the identity function $\varphi : V \to V : \varphi(v) = v, \forall v \in V$, which is an isomorphism from G to itself (cf. with proof of theorem 2.1).

Definition 2.59. (Cf. [64, p.15])(*Inverse permutation*) The *inverse permutation* ϕ^{-1} of a permutation $\phi = \begin{pmatrix} v_0 & v_1 & \cdots & v_n \\ \phi(v_0) & \phi(v_1) & \cdots & \phi(v_n) \end{pmatrix}$ of a graph with n vertices is defined by

$$\phi^{-1} = \begin{pmatrix} \phi(v_0) & \phi(v_1) & \cdots & \phi(v_n) \\ v_0 & v_1 & \cdots & v_n \end{pmatrix} \tag{34}$$

The above investigations encourage the following theorem.

Theorem 2.2. The set $Aut(G)$ of all graph automorphisms of a graph $G = (V, E)$ forms a group under function composition.

Proof. We proof the properties of a group[9].

(i) By definition, all permutations $\phi_n, \phi_k \in Aut(G)$ only perform structure preserving permutations. If a vertex $v \in V$ can not be swapped with another vertex of V without preserving structure, $\phi(v) = v = id(v), \forall \phi \in Aut(G)$.
Therefore $\phi_n \circ \phi_k, \forall \phi_n, \phi_k \in Aut(G)$ also only perform structure preserving permutations, hence $\phi_n \circ \phi_k \in Aut(G), \forall \phi_n, \phi_k$.

(ii) Suppose $\phi_1, \phi_2, \phi_3 \in Aut(G)$, then $\forall v \in V$ holds (cf. [49])

$$\phi_1 \circ (\phi_2 \circ \phi_3)(v) = \phi_1 \circ (\phi_2 \circ \phi_3(v)) = \phi_1 \circ \phi_2(\phi_3(v)) \quad = \phi_1(\phi_2(\phi_3(v))) \tag{35}$$
$$(\phi_1 \circ \phi_2) \circ \phi_3(v) = (\phi_1 \circ \phi_2)(\phi_3(v)) \ = (\phi_1 \circ \phi_2(\phi_3(v))) = \phi_1(\phi_2(\phi_3(v))) \tag{36}$$

Therefore, $\phi_1 \circ (\phi_2 \circ \phi_3) = (\phi_1 \circ \phi_2) \circ \phi_3$.

(iii) The identity id_G of G results in G, hence $id_G \in Aut(G)$.
Clearly $id_G \circ \phi = \phi \circ id_G = \phi, \forall \phi \in Aut(G)$.

[9]See definition A.9 in the appendix.

(iv) The inverse ϕ^{-1} of an automorphism ϕ is obviously an automorphism of G, thus $\phi^{-1} \in Aut(G)$. Since a permutation $\phi : V \to V$ is a bijection, ϕ^{-1} is also a bijection[10]. Therefore, $\exists v \in V : \phi(v') = v \Rightarrow \phi^{-1}(v) = v'$ with $v' \in V$, hence $\phi \circ \phi^{-1}(v) = \phi(\phi^{-1}(v)) = \phi(v') = v = id(v)$. Since the domain and the codomain, speaking of V, coincide, it holds that[11]

$$\phi \circ \phi^{-1}(v) = \phi^{-1} \circ \phi(v) = v = id(v), \forall v \in V \tag{37}$$

□

Definition 2.60. [43, p.3](*Symmetric group*) The group of all permutations of a graph G is called *symmetric group*, and is denoted by $Sym(G)$.

Corollary 2.5. Given a graph $G = (V, E)$. $Sym(G)$ is non-abelian for $|V| > 2$.

Proof. Assume $Sym(G)$ with $|V| \geq 3$ is abelian. Given two permutations $\phi, \phi' \in Sym(G)$ defined by

$$\phi = \begin{pmatrix} v_0 & v_1 & v_2 & \cdots & v_n \\ v_2 & v_0 & v_1 & \cdots & \phi(v_n) \end{pmatrix} \qquad \phi' = \begin{pmatrix} v_0 & v_1 & v_2 & \cdots & v_n \\ v_0 & v_2 & v_1 & \cdots & \phi(v_n) \end{pmatrix} \tag{38}$$

$$\phi \circ \phi' = \begin{pmatrix} v_0 & v_1 & v_2 & \cdots & v_n \\ v_2 & v_1 & v_0 & \cdots & \phi(v_n) \end{pmatrix} \qquad \phi' \circ \phi = \begin{pmatrix} v_0 & v_1 & v_2 & \cdots & v_n \\ v_1 & v_0 & v_2 & \cdots & \phi(v_n) \end{pmatrix} \tag{39}$$

It holds $\phi \circ \phi' \neq \phi' \circ \phi$, contradicting the assumption. □

Corollary 2.6. $Aut(G)$ is a subgroup of $Sym(G)$.

Proof. Since $Sym(G)$ contains all permutations of G, $Aut(G) \subseteq Sym(G)$. $Aut(G)$ satisfies the properties of a group (see proof of theorem 2.2). □

Notice that $Sym(G) = Aut(G)$, if G is complete [50]. Homans also proposes the following upper and lower bound.

Proposition 2.1. For a graph $G = (V, E)$, $1 \leq |Aut(G)| \leq |V|!$.

Proof. The identity permutation id_G is the trivial automorphism of G, hence the lower bound is 1. Since $Aut(G) \subseteq Sym(G)$, the upper bound is given by $|V|! = |Sym(G)|$, which is achieved if G is complete. □

In this section we have explored the concept of abstract graph isomorphism and examined the relationship between the graph theoretic approach and abstract algebra.

[10] See theorem A.1 in the appendix.
[11] Compare with the proof of lemma A.1 in the appendix for a permutation orientated approach.

In the work of Rodriguez a more in-depth knowledge of the group theoretic investigations of specific graph families is presented. He also gives an introduction to group theoretic automorphism and its connection to simple graphs. A similar point of view is presented in the work of Homans, who gives a deeper understanding on why graph automorphisms form a group under composition. Furthermore, in the work of Toomey a graph theoretic representation of a group, called *Cayley graph*, is presented. It appears that cayley graphs have many properties closely linked to the structure of a group.
Ghosh and Kurur study various permutation group algorithms and its complexity. They establish a connection between abstract graph isomorphism.

The topic of graph isomorphism is crucial for an efficient graph generation. Additionally, we are going to present a slightly different approach of the definition of graph isomorphism, as given by Brinkmann and McKay, while keeping planar embeddings and their rotation system in mind. The definition is used in the graph isomorphism detection program *nauty*, where a method called *canonical construction path* is implemented. The algorithm was developed by McKay.

2.3 Topology of planar graphs

In the last sections we had a natural intuition on what a *graph* is and how it can be, informally speaking, geometrically represented. We have agreed that a vertex is a point, and an edge is a line between two points. However, we have not given a precise idea on what a graph drawing actually is in a mathematical sense. In the following section, we are going to establish a connection between graph drawings and topology in order to give a definition on what a drawing of a graph is. Then, we will introduce a family of graphs called *planar graphs*. If one does not require mathematical precision, the idea of planarity is initially very intuitive to a certain limit. However, in most literature *planarity* and drawing of graphs in general is introduced with basic terms of topology. For this reasons we consider it a necessity to introduce basic topological terms, such as *topological space*, *embedding* or *homeomorphism*.
We are not so much concerned with theorems and formal proofs in this area, but much more we want to present the basic idea of topological graph theory.

Unlike most introductions to the topic of topology, we are not going to start with the definition of a topology or a topological space, but begin with the notion of a metric space, which is well known in (real) analysis.

Definition 2.61. [102, p.518](*Metric space*) A *metric space* is a nonempty set A together with a real-valued function ρ defined on $A \times A$ such that if u, v, and w are arbitrary members of A, then

(i) $\rho(u, v) \geq 0$ with equality if and only if $u = v$.

(ii) $\rho(u, v) = \rho(v, u)$.

(iii) $\rho(u, v) \leq \rho(u, w) + \rho(w, v)$.

We say that ρ is a *metric* on A.

Definition 2.62. [102, p.525](*ϵ-neighborhood*) Given a metric space A. If $u_0 \in A$ and $\epsilon > 0$, the set

$$N_\epsilon(u_0) = \{u \in A \mid \rho(u_0, u) < \epsilon\} \tag{40}$$

is called an *ϵ-neighborhood* of u_0. If a subset of S of A contains an ϵ-neighborhood of u_0, then S is a *neighborhood* of u_0, and u_0 is an interior point of S. The set of interior points of S is the *interior* of S, denoted by S^0.

Example 2.63. [102, p.521] Given the real coordinate space $\mathbb{R}^n$ with the cartesian coordinates $x = (x_1, x_2, \cdots, x_n)$ and $y = (y_1, y_2, \cdots, y_n)$. We define a metric $\rho_e(x, y)$ on $\mathbb{R}^n$ by

$$\rho_e(x, y) = \left(\sum_{i=1}^{n} (x_i - y_i)^2 \right)^{\frac{1}{2}} \tag{41}$$

The above metric is also known as the *euclidian distance* [102, p.283].

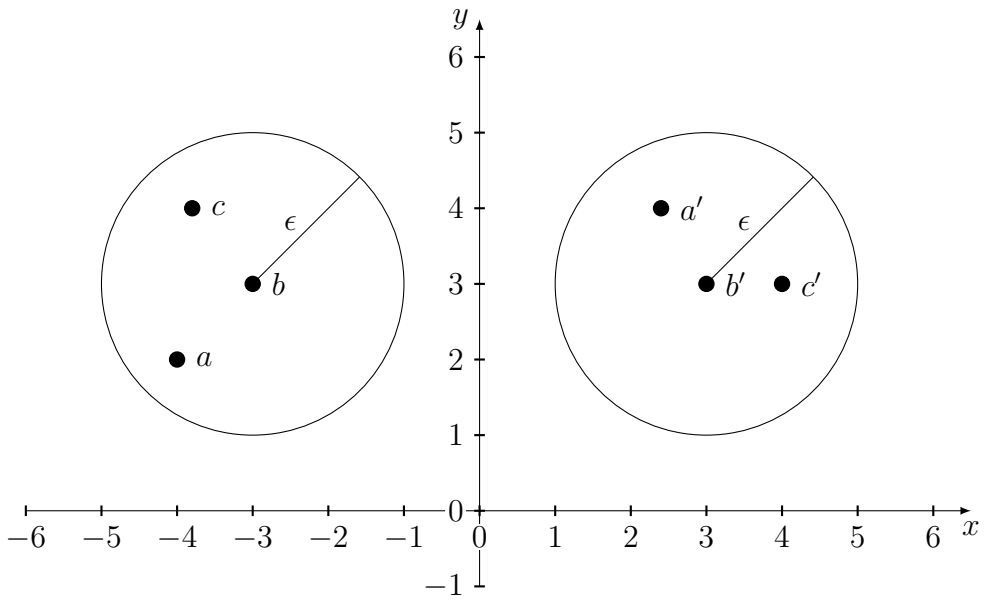


Figure 2.15: Two sets A and B of points in $\mathbb{R}^2$.

Given $\mathbb{R}^n$ with $n = 2$ and the euclidian metric ρ_e and two sets $A = \{a, b, c\}$ and $B = \{a', b', c'\}$ of points. We define two neighborhoods $S_b(b) = \{x \in \mathbb{R}^2 \mid \rho(b, x) < \epsilon\}$ and

$S_{b'}(b') = \{x \in \mathbb{R}^2 \mid \rho(b', x) < \epsilon\}$. Let $A \subseteq S_b(b)$ and $B \subseteq S_{b'}(b')$. The points of the sets A, B and the neighborhoods of b and b' are visualized in figure 2.15. We can see that the points of sets A and B are separated through their respective neighborhoods. In other words: the defined metric space formalizes our intuitive idea of what "distance" is.

Now, we want to generalize the concept of a metric space and the idea of a geometric "distance"-relation associated with it. In fact, we only want to use set theory. We might ask: "What is the minimal geometric relation that a set can be endowed with when we cannot [...] measure a notion of distance between elements?" [62]. This question motivates the definition of a *topology* on a given set.

Definition 2.64. [84, p.76](*Topology*) A *topology* on a set X is a collection $\mathcal{T}$ of subsets of X having the following properties:

(i) $\emptyset, X \in \mathcal{T}$.

(ii) The union of the elements of any subcollection of $\mathcal{T}$ is in $\mathcal{T}$.

(iii) The intersection of the elements of any finite subcollection of $\mathcal{T}$ is in $\mathcal{T}$.

Definition 2.65. [84, p.76](*Topological space, open sets*) Given a topology $\mathcal{T}$ on a set X. The ordered pair $(X, \mathcal{T})$ is called a *topological space*. The elements of $\mathcal{T}$ are called *open sets*.

Definition 2.66. [84, p.93](*Closed set*) A subset A of a topological space $(X, \mathcal{T})$ is said to be *closed* if the set $X - A$ is open.

Definition 2.67. [18, p.10](*Clopen set*) A subset A of a topological space X is called *clopen* if it is both open and closed in X.

If there can not occur any misunderstanding in the given context, we will call a topological space simply X.

Example 2.68. Given a set $X = \{a, b, c, a', b', c'\}$ and a collection $\mathcal{T} = \{X, \emptyset, \{a, b, c\}, \{a', b', c'\}\}$. Is $\mathcal{T}$ a topology on X? We presuppose that for any two sets Y, Z the following holds:

1. $Y \cup Y = Y$.
2. $Y \cap Y = Y$.
3. $Y \cup \emptyset = Y$.
4. $Y \cap \emptyset = \emptyset$.
5. If $Y \subseteq Z$, then $Y \setminus Z = \emptyset$.

We check the definition:

(i) Obviously, $\emptyset$ and X are in $\mathcal{T}$.

(ii) $\{a, b, c\} \cup \{a', b', c'\} = X \in \mathcal{T}$. Since $\{a, b, c\} \subset X$ and $\{a', b', c'\} \subset X$, their union with X is X.

(iii) $\{a, b, c\} \cap \{a', b', c'\} = \emptyset \in \mathcal{T}$. Since $\{a, b, c\} \subset X$ and $\{a', b', c'\} \subset X$, their intersection with X is $\{a, b, c\}$ and $\{a', b', c'\}$, respectively.

Therefore, $\mathcal{T}$ is a topology on X. It is visualized in figure 2.16. Each element of X is drawn as a point, each element of $\mathcal{T}$ as an ellipse around its elements.

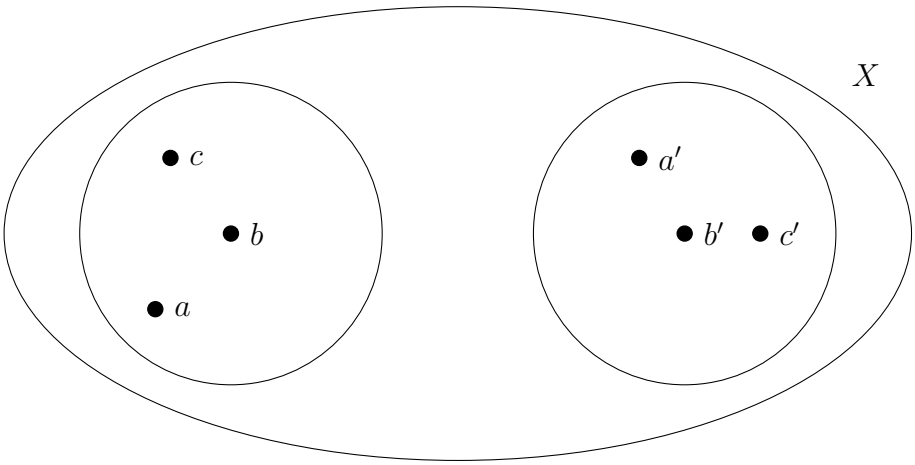


Figure 2.16: Example of a topology $\mathcal{T}$ on a given set X.

Even if it does not matter how we represent the elements of X and their set-relation, one can imagine that the relationship induced by $\mathcal{T}$ is very similar to the relationship we have noticed in example 2.63.

Definition 2.69. [18, p.4](*Neighborhood*) If $(X, \mathcal{T})$ is a topological space and $x \in X$ then a set N is called a *neighborhood* of x in X if there is an open set $U \subseteq N$ with $x \in U$. If $U = N$, the neighborhood N is called *open neighborhood* of x.

Example 2.70. Given the topological space X of example 2.68. Each open set is a neighborhood of all points it contains, e.g. the open set $\{a, b, c\}$ is the neighborhood of a, b and c. Notice that an neighborhood is not necessarily an open set, but all open sets are open neighborhoods of their points. Especially, the set X itself is open, and therefore a neighborhood of all given points of X [18, p.4].

Theorem 2.3. Let $(X, \mathcal{T})$ be a topological space, then $\emptyset$ and X are clopen.

Proof. $X, \emptyset \in \mathcal{T}$ by definition, hence they are open. But $X - X = \emptyset \in \mathcal{T}$ and $X - \emptyset = X \in \mathcal{T}$, hence $X, \emptyset$ are closed. Therefore, they are clopen. □

Definition 2.71. [84, p.148](*Separation*) Let X be a topological space. A *separation* of X is a pair (U, V) of disjoint nonempty open subsets of X whose union is X. The space X is said to be *connected* if there does not exist a separation of X.

Example 2.72. Given the topological space X of example 2.68. X is disconnected, since $\{a, b, c\} = A$ and $\{a', b', c'\} = B$ with $A \cap B = \emptyset$ are open and $\{a, b, c\} \cup \{a', b', c'\} = X$.

Definition 2.73. [84, p.103](*Continuous*) Let X and Y be topological spaces. A function $f : X \to Y$ is said to be *continuous* if for each open subset V of Y, the set $f^{-1}(V)$ is an open subset of X.

Definition 2.74. [84, p.105](*Homeomorphism*) Let X and Y be topological spaces. Let $\psi : X \to Y$ be a bijection. If both the function ψ and the inverse function

$$\psi^{-1} : Y \to X \tag{42}$$

are continuous, then ψ is called *homeomorphism*. We say that X and Y are *homeomorphic*, and write $X \cong Y$.

Notice that two topological spaces are considered identical if their exists an homeomorphism between them. Therefore, the term homeomorphism is the equivalent of the term graph isomorphism in the world of topology.

Definition 2.75. [84, p.105](*Topological invariant*) Given a homeomorphism $\psi : X \to Y$ between two topological spaces X and Y. Any property of X that is entirely expressed in terms of the topology of X (that is, in terms of the open sets of X) yields the corresponding property for the space Y. Such a property of X is called a *topological property* or *topological invariant* of X.

We can find topological invariants, which are inspired by the invariants we have found in the section about graph isomorphism, e.g. the number of open sets in $\mathcal{T}$ is a homeomorphic invariant.

Notice that a homeomorphism is an equivalence relation. The proof is similar to the one given for theorem 2.1.

Definition 2.76. [84, p.88](*Subspace topology*) Let X be a topological space with topology $\mathcal{T}$. If Y is a subset of X, the collection

$$\mathcal{T}_Y = \{Y \cap U \mid U \in \mathcal{T}\} \tag{43}$$

is a topology on Y, called the *subspace topology*. With this topology, Y is called subspace of X. Its open sets consist of all intersections of open sets of X with Y.

Lemma 2.3. [84, p.89] Let Y be a subspace of X. If U is open in Y and Y is open in X, then U is open in X.[12]

We have now introduced the topological foundations to be able to introduce planarity and graph drawings in general.

Definition 2.77. [32, p.90](*Straight line segment*) A *straight line segment* in the euclidian plane is a subset of $\mathbb{R}^2$ that has the form $L = \{p + \lambda(q - p) \mid 0 \leq \lambda \leq 1\}$ for distinct points $p, q \in \mathbb{R}^2$. Instead of *straight line segment*, we shall simply say *line segment*.

The above definiton is reminiscent of the vector equation of a line.

Example 2.78. Given two points $p, q \in \mathbb{R}^2$ and a scalar $\lambda \in \mathbb{R}$. The equation of a line through p and q is given by $\{p + \lambda(q - p) \mid \lambda \in \mathbb{R}\}$.

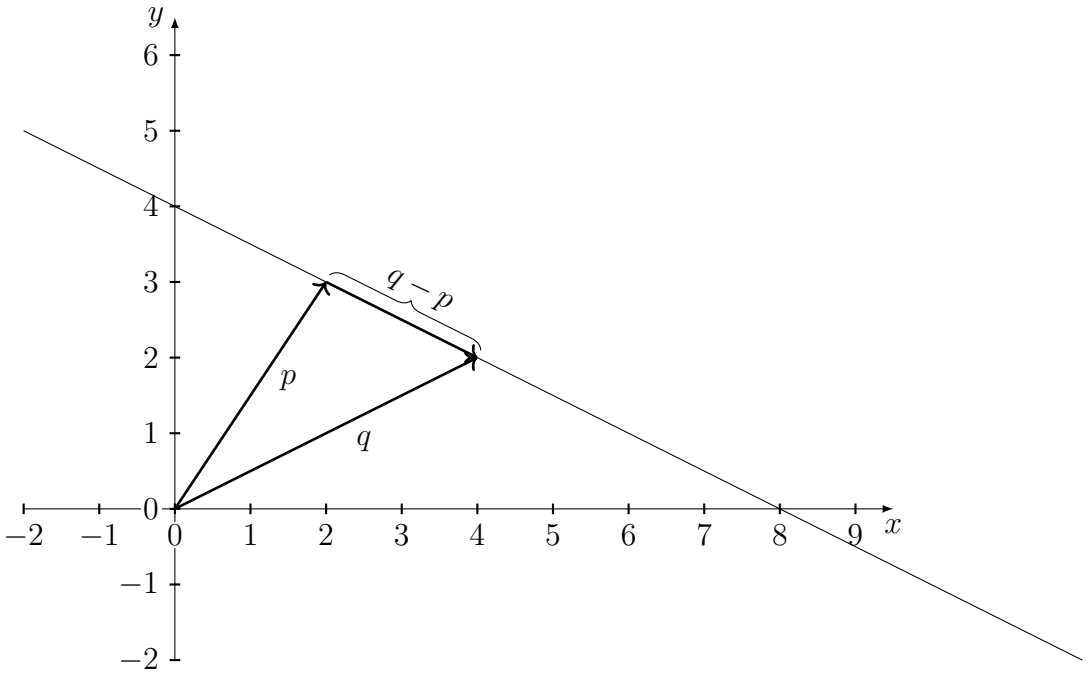


Figure 2.17: Vector equation of a line through two given points p and q.

The construction of the line with the given points p and q is shown in figure 2.17. Since λ is in $\mathbb{R}$ without restrictions, we reach all points on the line passing through p and q. Now, a straight line segment is a line, given by the equation above, limited to the length of $q - p$ (if we choose the upper bound for λ given by the definition).

Definition 2.79. [32, p.90](*Polygonal arc*) A *polygonal arc* is a subset of $\mathbb{R}^2$ which is the union of finitely many straight line segments and is homeomorphic to the closed unit interval $I_1 = [0, 1]$. The images of 0 and 1 under such a homeomorphism are the *endpoints* of this polygonal arc, which *links* them and *runs* between them. Instead of *polygonal arc* we shall simply say *arc*.

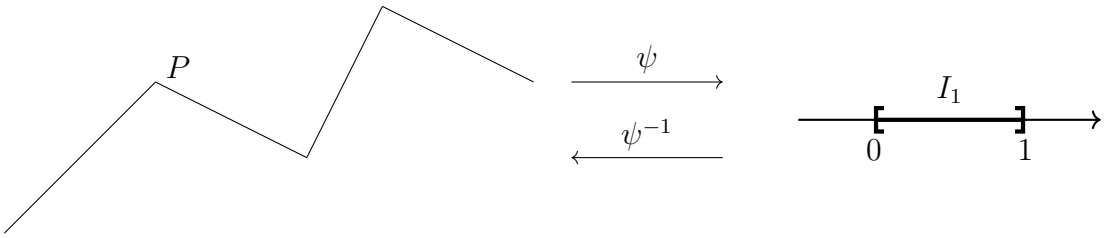


Figure 2.18: A polygonal arc is homeomorphic to the unit interval.

In figure 2.18 an example for an arc P is given. We show that P is homeomorphic to the unit interval. First, we show homeomorphism for a single line segment. We define the homeomorphism by

$$\psi : [0,1] \to L : \lambda \mapsto p + \lambda(q-p),\ p,q \in \mathbb{R}^2,\ p \neq q \tag{44}$$

Then for $\lambda = 0$, we have $0 \mapsto p$. For $\lambda = 1$, we have $1 \mapsto p+q-p = q$. The upper and lower bound of the interval map to the endpoints of the segment. We define the inverse mapping by[13]

$$\psi^{-1} : L \to [0,1] : p + \lambda(q-p) \mapsto \lambda,\ p,q \in \mathbb{R}^2,\ p \neq q,\ 0 \leq \lambda \leq 1 \tag{45}$$

For $\lambda = 0$, we have $p \mapsto 0$. For $\lambda = 1$, we have $q \mapsto 1$. The endpoints of the segment map to the upper and lower bound of the interval. The function ψ defines a homeomorphism between L and $[0,1]$. For a polygonal arc with more than one line segment the construction of the homeomorphism is similar.

Definition 2.80. [32, p.90](*Arc interior*) If P is an arc between two points x and y, we denote the point set $P \setminus \{x,y\}$, called the *interior* of P, by $\mathring{P}$.

Example 2.81. Given two points $x, y \in \mathbb{R}^2$. Let P be an arc between x and y, then all points in P with the exception of x and y are the interior of P. The arc P is visualized in figure 2.19. The curly bracket indicates the interior set $\mathring{P}$ of P. The points x and y are visualized as black dots.

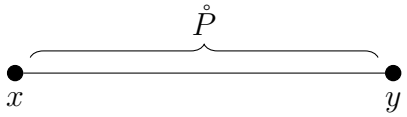


Figure 2.19: The interior $\mathring{P}$ of an arc P.

[12]See [84, p.89] for a proof.
[13]The inverse was found with the help of Mosher and Ryffel.

Definition 2.82. [32, p.90](*Polygon*) A *polygon* is a subset of $\mathbb{R}^2$, which is the union of finitely many straight line segments and is homeomorphic to the unit circle S^1.[14]

Definition 2.83. [32, p.90](*Region*) Let $\mathcal{O} \subseteq \mathbb{R}^2$ be any open set. Being linked by an arc in $\mathcal{O}$ defines an equivalence relation on $\mathcal{O}$. The corresponding equivalence classes are again open. They are the *regions* of $\mathcal{O}$. We denote the relation by $\frown$.

Notice: we say that $x \frown y, \forall x, y \in \mathcal{O}$ "defines" an equivalence relation on $\mathcal{O}$, hence reflexivity, meaning $x \frown x, \forall x \in \mathcal{O}$, is explicitly assumed.[15]

Definition 2.84. [32, p.90](*Separation*) A closed set $X \subseteq \mathbb{R}^2$ is said to *separate* a region $\mathcal{O}'$ of $\mathcal{O}$ if $\mathcal{O}' \setminus X$ has more than one region.

Definition 2.85. [32, p.90](*Frontier*) The *frontier* of a set $X \subseteq \mathbb{R}^2$ is the set Y of all points $y \in \mathbb{R}^2$ such that every neighborhood of y meets both X and $\mathbb{R}^2 \setminus X$. Note that if X is open then its frontier lies in $\mathbb{R}^2 \setminus X$.

Definition 2.86. [32, p.92](*Plane graph*) A *plane graph* is a pair (V, E) of finite sets with the following properties (the elements of V are again called *vertices*, those of E *edges*):

(i) $V \subseteq \mathbb{R}^2$.

(ii) Every edge is an arc between two vertices.

(iii) Different edges have different sets of endpoints.

(iv) The interior of an edge contains no vertex and no point of any other edge.

The abstract graph of a plane graph is called *planar* graph. As long as no confusion arises, we shall use the same name for the abstract graph and its plane graph.

Definition 2.87. (Cf.[32, p.92])(*Face*) When G is a plane graph, we call the regions of $\mathbb{R}^2 \setminus G$ the *faces* of G. These are open subsets of $\mathbb{R}^2$ and hence have their frontiers in G. Since G is bounded, exactly one of its faces is unbounded. This face is the *outer face* of G. The other faces are its *inner faces*. We denote the set of faces of G by $F(G)$.

Definition 2.88. [32, p.98](*Planar embedding*) An embedding in the plane, or *planar embedding*, of an abstract graph G is an isomorphism between G and a plane graph $\widetilde{G}$. The latter will be called a *drawing* of G. We shall not always distinguish notationally between the vertices and edges of G and $\widetilde{G}$.

[14] See [94] for a proof.

[15] In a private e-mail from [31].

Example 2.89. Given a plane graph $G = (V, E)$ with $V = \{v_0, v_1, v_2\}$ and $E = \{(v_0, v_1), (v_0, v_2), (v_1, v_2)\}$. G defines K_3, which is a triangle. Define $X = \mathbb{R}^2 \setminus G$, X is open in $\mathbb{R}^2$. X is the union of two disjoint subsets $I = \mathbb{R}^2 \setminus (G \cup O)$ and $O = \mathbb{R}^2 \setminus (G \cup I)$. They are the regions of X and are by definition again open in $\mathbb{R}^2$. To summarize, we define a topology on $\mathbb{R}^2$ by $\mathcal{T} = \{\mathbb{R}^2, \emptyset, X, I, O\}$. Notice that G is closed, since X is open. $I \cap O = \emptyset \in \mathcal{T}$, $I \cup O = X \in \mathcal{T}$ and implicitly $I \cup X = X, O \cup X = X, X \cap I = I, X \cap O = O$. In figure 2.20 the open sets X, I, O and $\mathbb{R}^2$ as well as the plane graph G are visualized. One can see that no point in I can be connected with any point in O by an arc without intersecting the interior of an edge of G or without intersecting a vertex of G.

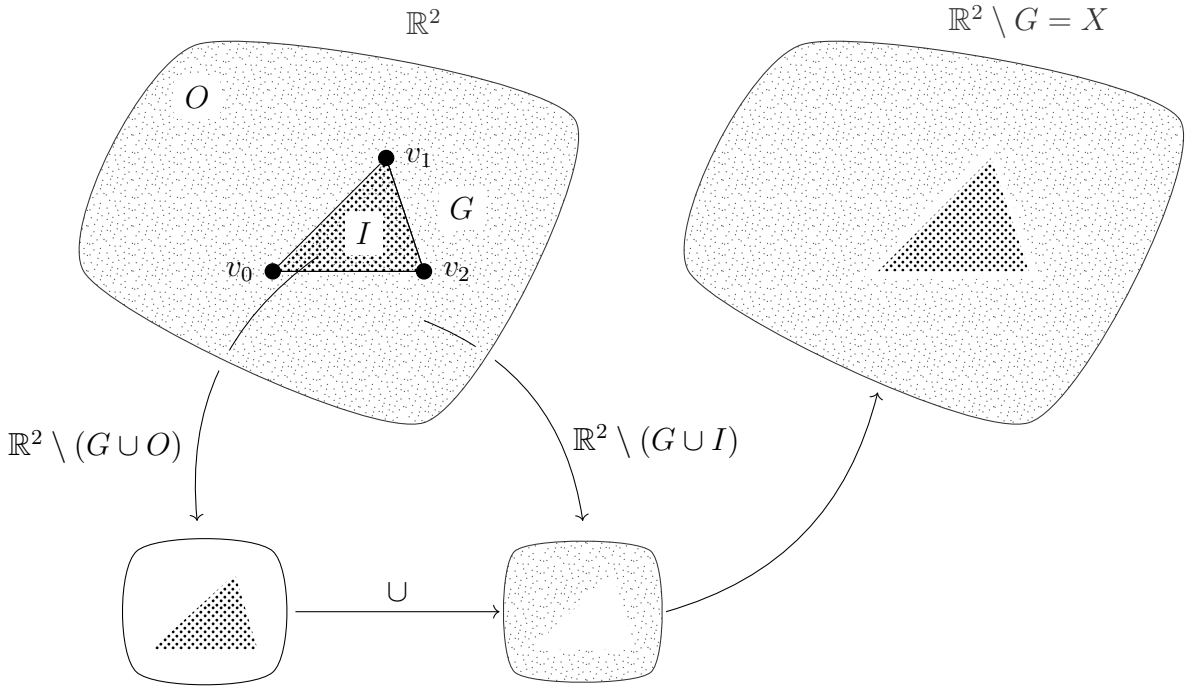


Figure 2.20: Embedding K_3 in $\mathbb{R}^2$, whereby $\mathbb{R}^2 \setminus K_3$ has exactly two regions, the faces of K_3.

Notice that in $\mathbb{R}^2 \setminus G = X$, the plane is separated into two disjoint regions such that we cannot find a continuous map (e.g. an arc) which connects any point of I with any point of O (compare with the figure on the right side). They form equivalence classes under $\frown$. The concept of a region is very similar to the definition of an *arc-component* as given in the work of Munkres, which is also an equivalence class under arc connection. In that sense, a face is an arc-component. This concept can be generalized by introducing the term *path-connected*. See [84] for more information.

Furthermore, the above results also explain that one edge alone does not separate the plane, or more generally speaking, if no subset of $\mathbb{R}^2$ is bounded by a cycle of G, which is embedded as a polygon, the only face of G is the outer face. We often call the set I of G the *interior* of G (more precisely the interior of a face of G) and the set O (the outer face) the *exterior* of G.

The investigations, which we have made, are formalized in the *Jordan Curve Theorem*, which is well known in literature. Further information as well as formal proofs can be found in [32, 45, 84].

Leonhard Euler conjectured a connection between the vertices, edges and the faces of planar graphs. It is known as the *Euler characteristic*, sometimes *Euler's formula*[16], and was originally defined for polyhedra [8].

Theorem 2.4. Let G be a connected plane graph with n vertices, m edges, and f faces. Then

$$\mathcal{X} = n - m + f = 2 \tag{46}$$

$\mathcal{X}$ defines a topological invariant [8]. Several proofs are known in literature; see [32, p.97] for an example.

Definition 2.90. [32, p.110](*Dual graph, plane dual*) Given two plane graphs $G = (V, E)$ and $G^* = (V^*, E^*)$ and put $F(G) := F$ and $F(G^*) = F^*$. We call G^* a *plane dual* of G, if there are bijections

$$\begin{array}{lll} F \to V^* & E \to E^* & V \to F^* \\ f \mapsto v^*(f) & e \mapsto e^* & v \mapsto f^*(v) \end{array} \tag{47}$$

satisfying the following conditions:

(i) $v^*(f) \in f, \forall f \in F$.

(ii) $|e^* \cap G| = |\mathring{e}^* \cap \mathring{e}| = |e \cap G^*| = 1, \forall e \in E$ and in each of e and e^* this point is an inner point of a straight line segment.

(iii) $v \in f^*(v), \forall v \in V$.

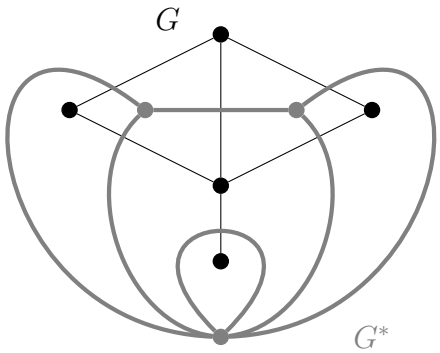


Figure 2.21: The plane dual G^* of G [32].

[16]Not to be mistaken with Euler's formula in complex analysis, known as $e^{ix} = cos\, x + i\, sin\, x$.

Example 2.91. Given a plane graph $G = (V, E)$ with 5 vertices and 6 edges. G and its plane dual G^* are visualized in figure 2.21. Notice that in every face of G there is a vertex of its dual G^*.

In this section we have introduced topological graph theory in the context of planar graphs. We presented basic terms of topology, such as *topological space*, *homeomorphism* or *region*. Then, we introduced plane graphs with the terms of polygonal arcs, polygons and showed homeomorphism to the unit interval, to the unit circle, respectively. The given introduction to the topic was indeed very brief and we could not cover many important aspects of topological graph theory. The reader is referred to the work of Munkres for an all-encompassing introduction to the field of topology. However, he is more concerned with general topology than with topological graph theory. Additionally his work might be too complex for a brief overview. Therefore, the reader is also referred to [80, 45, 96] and [94].

2.4 Rotation systems

In the last section we gave a topological definition of plane graphs. However, if one want to generate graphs and their embeddings, the topological definition does not seem to match the expectations, when considering an implementation in an actual program. We might ask how we can construct a plane graph for a given planar graph. The computational problem, implied by this question, is studied in the research area called *graph drawings*. Despite the theoretical interest, it has great practical impact, e.g. in biochemistry, network visualization and web site traffic analysis [63, 30].

An embedding of a graph can be described with a combinatorial object, called *rotation system*.

Definition 2.92. [80, p.90](*Rotation system*) Let $G = (V, E)$ be a connected graph. Let $\Pi = \{\pi_v \mid v \in V\}$ where π_v is the cyclic permutation of the edges incident with the vertex v such that $\pi_v(e)$ is the successor of e in the clockwise ordering around v. The cyclic permutation π_v is called the *local rotation* at v, and the set Π is the *rotation* system of the given embedding of G.

Definition 2.93. [80, p.100](Π-*consecutive,* Π-*angle*) Edges e and e' incident with vertex v are Π-*consecutive* if $\pi_v(e) = e'$ or $\pi_v(e') = e$. Every such pair $\{e, e'\}$ of edges forms a Π-*angle*.

Example 2.94. Given a graph $G = (V, E)$ with $V = \{v_0, v_1, v_2, v_3, v_4\}$ and $E = \{(v_0, v_2), (v_0, v_3), (v_2, v_1), (v_2, v_3), (v_2, v_4), (v_1, v_4), (v_3, v_4)\}$. One possible embedding is visualized in figure 2.22.

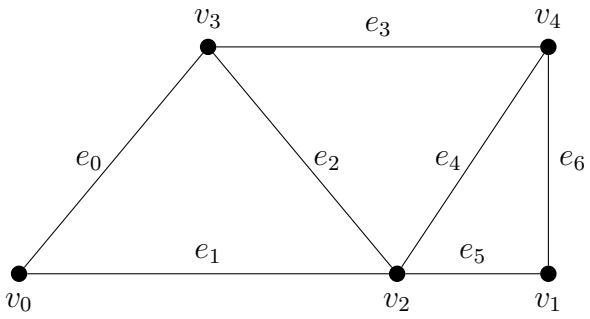


Figure 2.22: Embedding of a graph G according to the given rotation system Π_G (cf. [19]).

The given embedding can be encoded with a rotation system $\Pi_G = \{\pi_{v_0}, \pi_{v_1}, \pi_{v_2}, \pi_{v_3}, \pi_{v_4}\}$. Since we have choosen clockwise ordering around the vertices, the local rotations are given by

$$\pi_{v_0} = \begin{pmatrix} e_0 & e_1 \\ e_1 & e_0 \end{pmatrix} \qquad \pi_{v_1} = \begin{pmatrix} e_5 & e_6 \\ e_6 & e_5 \end{pmatrix} \qquad \pi_{v_2} = \begin{pmatrix} e_1 & e_2 & e_4 & e_5 \\ e_2 & e_4 & e_5 & e_1 \end{pmatrix} \tag{48}$$

$$\pi_{v_3} = \begin{pmatrix} e_0 & e_2 & e_3 \\ e_3 & e_2 & e_0 \end{pmatrix} \qquad \pi_{v_4} = \begin{pmatrix} e_3 & e_4 & e_6 \\ e_6 & e_3 & e_4 \end{pmatrix} \tag{49}$$

For example, taking vertex v_2 and its local rotation π_{v_2}, inducing the cycle $e_1 \mapsto e_2 \mapsto e_4 \mapsto e_5 \mapsto e_1$, we are able to reconstruct v_2 with all edges incident with v_2. In figure 2.23 the local rotation at v_2 is visualized.

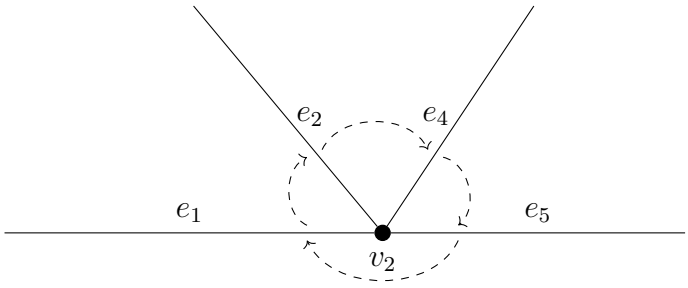


Figure 2.23: Local rotation π_{v_2} at v_2.

Definition 2.95. [80, p.91](*Rotation system equivalence*) Given a graph $G = (V, E)$ and two rotation systems $\Pi = \{\pi_v \mid v \in V\}$ and $\Pi' = \{\pi'_v \mid v \in V\}$. Π and Π' are *equivalent* if they are either the same or for each $v \in V$ we have $\pi'_v = \pi_v^{-1}$.

The concept of rotation systems can be generalized in so called *combinatorial maps*, which is a algebraic representation of a *topological map*. Jones and Singerman develop a category of topological maps and show how they can be described by a permutation group.

In the work of Zeps and Ķikusts it is shown how one get the actual drawing from a combinatorial map and in the work of Lando and Zvonkin an introduction to topological maps are given.

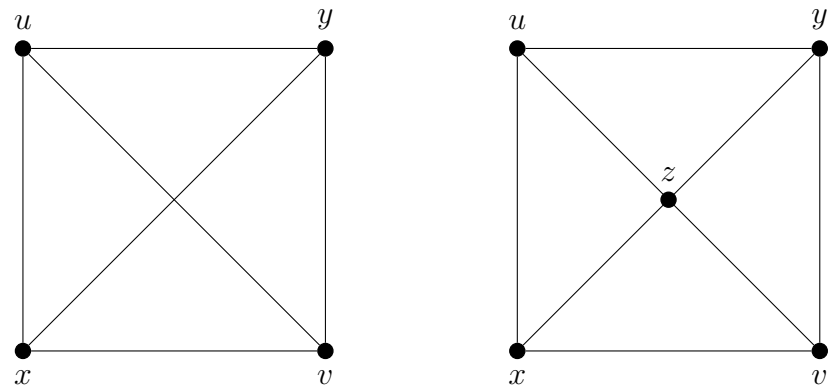
2.5 1-planar graphs

In the following we are going to give a brief overview of a generalization of planar graphs called 1-*planar graphs*. We are especially interested in certain subfamilies of 1-planar graphs such as *optimal* and *outer* 1-planar graphs. Unless otherwise stated, the following definitions and theorems are taken from a paper submitted by Fabrici and Madaras. In the following we only consider connected simple graphs, unless otherwise stated.
Generally speaking, a graph is called 1-planar "if it can be drawn in the plane so that each edge is crossed by at most one other edge" [33]. Therefore, every planar graph is also 1-planar. The notion of 1-planar graphs was introduced by Ringel.

Definition 2.96. [33, p.855](*Crossing*) Let G be a 1-planar graph and let $D(G)$ be a 1-planar diagram of G, that is, a drawing of G in which every edge is crossed by at most one other edge. If two arcs $x \frown y, u \frown v$ cross in $D(G)$ and z is their intersection, we say that z is the crossing of the edges $(x, y), (u, v)$.

Definition 2.97. [33, p.855](*Crossing vertices*) Let $C = C(D(G))$ be the set of all crossings in $D(G)$ and let E_0 be the set of all non-crossed edges in $D(G)$. The *associated plane graph* $D(G)^\times$ of $D(G)$ is the plane graph such that $V(D(G)^\times) = V(D(G)) \cup C$ and $E(D(G)^\times) = E_0 \cup \{(x, z), (y, z) \mid (x, y) \in E(D(G)) - E_0, z \in C, z \in (x, y)\}$. Thus, in $D(G)^\times$, the crossings of $D(G)$ become new vertices of degree 4; we call these vertices *crossing vertices*.

Example 2.98. Given an abstract graph $G = (V, E)$ with $V = \{x, y, u, v\}$ and $E = \{(x, v), (x, u), (x, y), (u, y), (u, v), (v, y)\}$.



(a) $D(G)$ of G. (b) $D(G)^\times$ of G.

Figure 2.24: The 1-planar diagram and the associated plane graph of G.

Let $D(G)$ be the 1-planar diagram of G, given by figure 2.24a, and $D(G)^\times$ the associated plane graph of G, given by figure 2.24b.
The set $C = C(D(G)) = \{z\}$ is the set of all crossings in $D(G)$. The set of vertices of $D(G)^\times$ is given by $V(D(G)^\times) = V \cup C = \{x, y, u, v, z\}$. In this example: $|C| = 1$.
Let $E_0 = \{(x, v), (v, y), (u, y), (x, u)\}$ be the set of all non-crossed edges. Then,

$$E(D(G)^\times) = E_0 \cup \{(x, z), (y, z) \mid (x, y) \in E(D(G)), z \in C, z \in (x, y)\} \tag{50}$$
$$= E_0 \cup \{(x, z), (u, z), (y, z), (v, z)\} \tag{51}$$

The degree of $z \in C$ is 4, satisfying definition 2.97.

Definition 2.99. [33, p.855](*Pseudo-outerface*) Let $B^\times$ be the boundary walk of the outerface of $D(G)^\times$ and let B the corresponding closed curve in $D(G)$. The *pseudo-outerface* of $D(G)$ is the set $Ext(B)$. We say that x is incident with the pseudo-outerface of $D(G)$ if x is a vertex of $D(G)$ which is incident with the outerface of $D(G)^\times$.

Notice, that $Ext(B)$ is the *exterior* of B (compare with section 2.3).

Example 2.100. The vertices x, y, v and u of $D(G)$ are incident with the pseudo-outerface of $D(G)$, since they are incident with the outerface of $D(G)^\times$ (compare with figure 2.24).

In section 2.4 we introduced a combinatorial object for representing planar embeddings, called *rotation system*. We generalize this idea for 1-planar graphs.

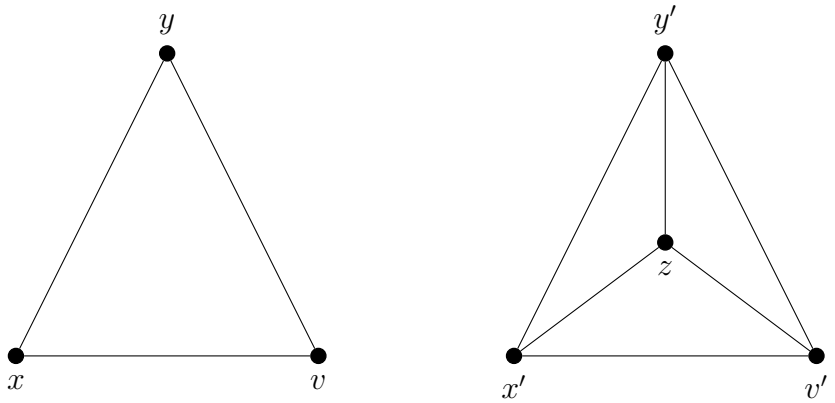
Definition 2.101. [3, p.69](*Planar, 1-planar rotation system*)
Given a graph G and a rotation system Π for G. We call Π a *planar rotation system* if it admits a plane drawing of G. We call Π a 1-*planar rotation system* if it admits a 1-plane drawing of G.

Auer et al. point out that "a 1-planar rotation system does not uniquely define a 1-planar embedding nor does it determine the pairs of crossing edges". They show that deciding whether a given rotation system is 1-planar is $\mathcal{NP}$-hard [3].

Next, we are going to present a subfamily of planar graphs, called *outerplanar* graphs, then we are going to generalize this idea once again to a subfamily of 1-planar graphs, called *outer* 1-planar graphs.

Definition 2.102. [4, p.2](*Outerplanar* graph) A planar graph G is called *outerplanar* if all vertices of G are in the outer face of G.

Example 2.103. Given two graphs $G = (V, E)$ with $V = \{x, v, y\}$ and $E = \{(x, v), (x, y), (y, v)\}$ and $G' = (V', E')$ with $V' = \{x', v', y'\}$ and $E' = \{(x', v'), (x', y'), (y', v'), (x', z), (v', z), (y', z)\}$. The graph G is outerplanar, since all vertices are in the outer face. G' is non-outerplanar, since z is not in the outer face of G'.



(a) Outerplanar graph G.

(b) Non-outerplanar graph G'.

Figure 2.25: Example of an outerplanar graph G and a non-outerplanar graph G'.

Definition 2.104. [4, p.3](*Outer 1-planar graph*) A graph G is called *outer 1-planar* if all vertices of G are in the outer face of G and each edge of G has at most one crossing.

Definition 2.105. [4, p.4](*Maximal outer 1-planar*) A graph G is *maximal* outer 1-planar if the addition of any edge violates outer 1-planarity.

Auer et al. state that every outer 1-planar graph with n vertices has at most $\frac{5}{2}n - 4$ edges. Maximal outer 1-planar graphs with n vertices have at most $\frac{11}{5}n - \frac{18}{5}$ edges.

They develop a linear-time algorithm for the efficient recognition of outer 1-planar graphs by using a data structure called *SPQR-trees*.

3 Optimal 1-planarity

In the following section we are going to focus on a subfamily of 1-planar graphs called *optimal* or *maximal* 1-planar graphs. We are interested in generating optimal 1-planar graphs, and then use them as an input for automated conjecture-making. For that purpose, we are going to investigate the structure of optimal 1-planar graphs in more detail.

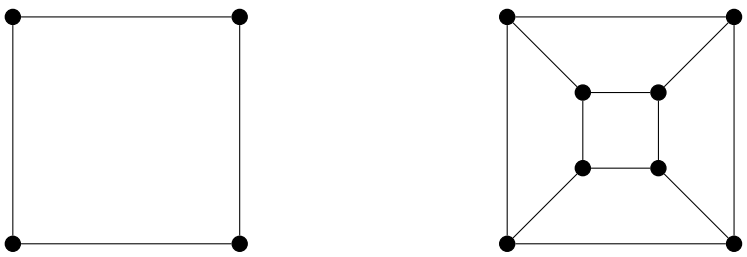
In the work of Bodendiek et al. it is proven, that optimal 1-planar graphs with n vertices only exist for $n = 8$ and $n \geq 10$. They also showed that there are no such graphs for $n \leq 7$ and $n = 9$. In the following we are going to investigate their results in more detail and give a precise definition of optimal 1-planar graphs. Especially, we will see that the underlying graph structure of optimal 1-planar graphs are characterized by simple quadrangulations. We will use this observation as a basis for our generation algorithm, which we will develop in a later section.

3.1 Simple quadrangulations

Definition 3.1. [22, p.34](*Simple quadrangulation*) A *quadrangulation* of the sphere is a finite graph embedded on the sphere, such that every face is bounded by a walk of 4 edges. A *simple quadrangulation* is a quadrangulation with no multiple edges. In other words: the boundary of each face of a simple quadrangulation corresponds to a 4-cycle of the graph. We sometimes shall simply write *quadrangulation* instead of *simple quadrangulation* if no confusion arises.

Less formally, each face of a simple quadrangulation G_Q corresponds to a subgraph G'_Q of G_Q, such that G'_Q can be embedded in $\mathbb{R}^2$ as a quadrilateral (polygon with 4 edges and 4 vertices [106]).

Example 3.2. In figure 3.1 and example of two simple quadrangulations is shown.



(a) 4 vertices. (b) 8 vertices.

Figure 3.1: Examples of simple quadrangulations of the sphere.

The quadrangulation in 3.1a has 4 vertices. The quadrangulation in 3.1b has 8 vertices. Notice how every face is bounded by a 4-cycle and how each face corresponds to a quadrilateral.

In the following we are first going to investigate some properties of simple quadrangulations. Then, we are going to give a definition of optimal 1-planar graphs, which are composed of simple quadrangulations.

Lemma 3.1. Given a simple quadrangulation $Q = (V, E)$, then for the average degree holds $\frac{1}{|V|} \sum_{v \in V} d(v) < 4$ [22].

Proof. We denote $|E|$ as the number of edges of Q, $|V|$ as the number of vertices of Q, and $|F|$ as the number of faces of Q. Assume that $\frac{1}{|V|} \sum_{v \in V} d(v) \geq 4$.

$$\Rightarrow \quad \sum_{v \in V} d(v) \geq 4|V| \tag{52}$$

$$\Leftrightarrow \quad 2|E| \geq 4|V| \qquad \text{(handshaking lemma)} \tag{53}$$

$$\Leftrightarrow \quad \frac{1}{2}|E| \geq |V| \tag{54}$$

Using Euler's formula: $|V| - |E| + |F| = 2$ and equation (54).

$$\tag{55}$$

$$\Rightarrow \quad \frac{1}{2}|E| - |E| + |F| \geq 2 \tag{56}$$

$$\Leftrightarrow \quad |F| - \frac{1}{2}|E| \geq 2 \tag{57}$$

$$\Leftrightarrow \quad |F| \geq 2 + \frac{|E|}{2} \tag{58}$$

Since every quadrangulation is bounded by a 4-cycle, meaning enclosed by 4 edges, each edge is used in 2 faces.

$$\Rightarrow \quad |E| = \frac{4}{2}|F| \qquad \Leftrightarrow \quad |F| = \frac{1}{2}|E| \tag{59}$$

And therefore

$$\frac{1}{2}|E| \geq 2 + \frac{|E|}{2} \tag{60}$$

A contradiction. □

Corollary 3.1. The connectivity of a simple quadrangulation cannot be more than 3.

Proof. Given a simple quadrangulation $Q = (V, E)$ with $|V| = n$. Assume that for each $v \in V$, $d(v) = 4$ holds, then

$$\frac{1}{|V|} \sum_{v \in V} d(v) = \frac{4n}{n} = 4 \tag{61}$$

A contradiction to lemma 3.1. Hence, there must be a vertex $v' \in V$ with $d(v') \leq 3$. The neighborhood of v' form a cutset. Therefore, for the connectivity k of Q holds $k \leq 3$. □

Lemma 3.2. If G is a plane graph, then G is 2-connected if and only if every face of G is bounded by a cycle [29].

The proof can be found in [29] and will not be repeated at this point. It follows from lemma 3.2 that a simple quadrangulation is at least 2-connected.

Let a graph G be k-connected with minimum degree $\delta(G)$. It is known that $k \leq \delta(G)$ (cf. [32, p.12]), hence $2 \leq \delta(Q)$ for a simple quadrangulation $Q = (V, E)$ according to lemma 3.2, and therefore $2 \leq \frac{1}{|V|} \sum_{v \in V} d(v)$.

Summarizing lemma 3.2, and, as a consequence of lemma 3.1, corolarry 3.1, the connectivity of a simple quadrangulation is either 2 or 3.

Corollary 3.2. Given a simple quadrangulation $Q = (V, E)$. The minimum degree $\delta(Q)$ of Q is 2 or 3 [22, p.39].

Proof. Since Q is at least 2-connected, $2 \leq \delta(Q)$. Using lemma 3.1:

$$2 \leq \delta Q \leq \frac{1}{|V|} \sum_{v \in V} d(v) \leq 3 \tag{62}$$

$$\Rightarrow \quad 2 \leq \delta Q \leq 3 \tag{63}$$

□

Lemma 3.3. Every 3-connected quadrangulation contains at least 8 vertices of degree 3.

The proof of the above lemma can be found in [10, p. 45].

By definition, a simple quadrangulation Q only contains cycles of even length. It is well known that graphs with no odd cycles are bipartite; a proof can be found in [14]. Therefore, Q is bipartite (cf. [12, p.325] and [22, p.39]).

Example 3.3. Given a simple quadrangulation $Q = (V, E)$ with $V = \{v_0, v_1, v_2, v_3, v_0', v_1', v_2', v_3'\}$ and $E = \{(v_0, v_1), (v_0, v_3), (v_0, v_0'), (v_2, v_1), (v_2, v_3), (v_2, v_2'), (v_1', v_1), (v_1', v_0'), (v_1', v_2'), (v_3', v_3), (v_3', v_2'), (v_3', v_0'), \}$. The plane graph of Q is visualized in figure 3.2.

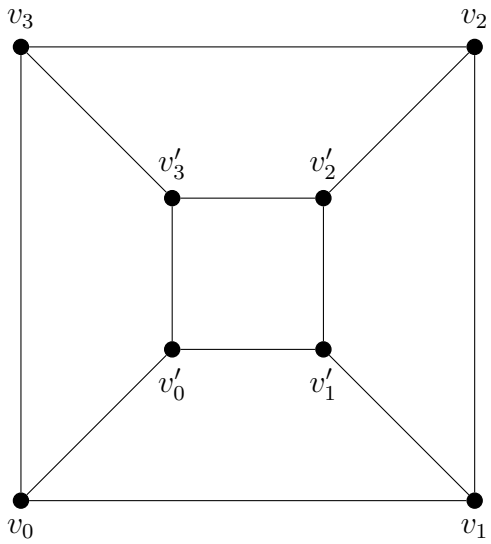


Figure 3.2: Simple quadrangulation Q with $n = 8$ vertices.

Q contains 6 cycles of length 4: $C_0 = v_0v_1v_2v_3v_0$ with length $k_0 = 4$, $C_1 = v_0v_1v_1'v_0'v_0$ with length $k_1 = 4$, $C_2 = v_1v_1'v_2'v_2v_1$ with length $k_2 = 4$, $C_3 = v_2v_2'v_3'v_3v_2$ with length $k_3 = 4$, $C_4 = v_0v_0'v_3'v_3v_0$ with $k_4 = 4$ and $C_5 = v_0'v_1'v_2'v_3'v_0'$ with length $k_5 = 4$. Furthermore, Q is bipartite. Hence, the vertex set V of Q can be partitioned into two sets V_1, V_2 with $V_1 \subset V$, $V_2 \subset V$ and $V_1 \cap V_2 = \emptyset$ and $V_1 \cup V_2 = V$. Furthermore, all vertices $v \in V_1$ are non-adjacent and all vertices $v' \in V_2$ are non-adjacent. The partitions are given by $V_1 = \{v_0, v_2, v_1', v_3'\}$ and $V_2 = \{v_1, v_3, v_0', v_2'\}$. The bipartition is visualized in figure 3.3.

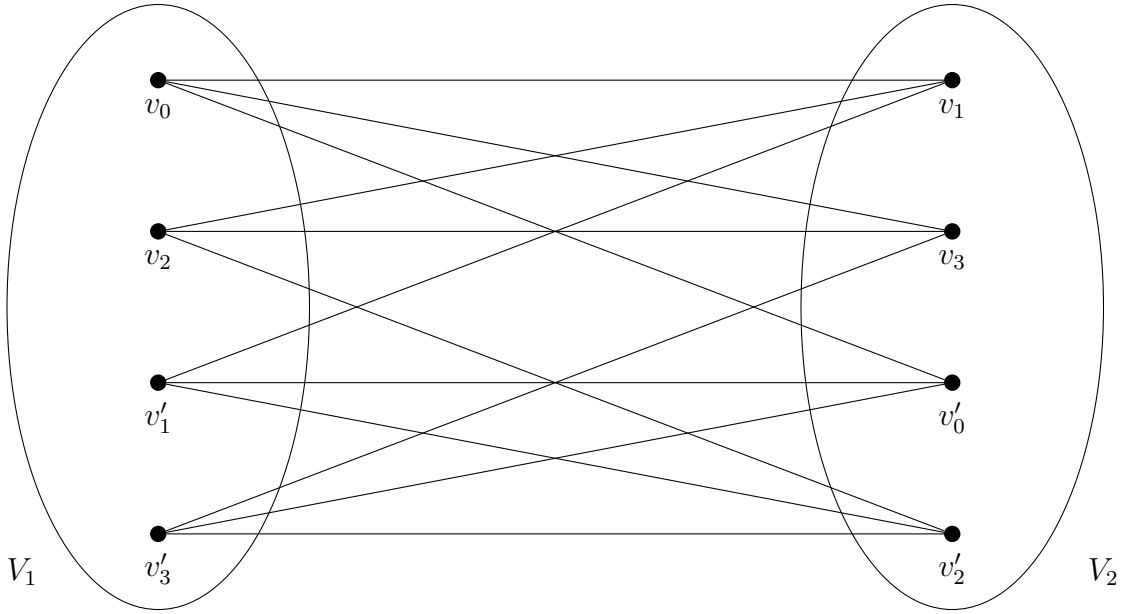


Figure 3.3: Bipartition of Q.

Each vertex $v \in V_1$ and each vertex $v' \in V_2$ has degree 3. The above bipartition is going to be useful for some of the properties we are going to derive for 1-planar graphs.

3.2 Structure and properties of optimal 1-planar graphs

Before we introduce the term *optimal* for 1-planar graphs, we need the following corollary, which is proven in [32].

Corollary 3.3. A plane graph G with $n \geq 3$ vertices has at most $3n - 6$ edges. Every plane triangulation with n vertices has $3n - 6$ edges.

If a plane graph G has $3n - 6$ edges, then we call G *optimal* or *maximal*.

Lemma 3.4. Let G be a 1-planar graph on n vertices and m edges. Then $m \leq 4n - 8$.

The following proof is taken from [33], and shown here again, since it is important for the next steps.

Proof. Consider the maximal 1-planar graph G on n vertices and let $D(G)$ be a 1-planar diagram of G. Let c be the number of crossings in $D(G)$. If two edges $(x, y), (z, w) \in E(G)$ mutually cross in $D(G)$, then $(x, z), (x, w), (y, z), (y, w) \in E(G)$ by the maximality of G. Therefore, if we delete an edge from each pair of edges which are crossing in $D(G)$, the resulting graph G' is a plane triangulation on $n' = n$ vertices, m' edges and f' faces. By Euler's theorem $m' \leq 3n' - 6$ (corollary 3.3) and therefore

$$m' - n' + f' \leq 2 \tag{64}$$

$$3n' - 6 - n' + f' \leq 2 \tag{65}$$

$$f' \leq 2n' - 4 \tag{66}$$

Now, $m = m' + c \leq m' + \frac{f'}{2} \leq 3n - 6 + n - 2 = 4n - 8$. □

If a graph G with n vertices has $4n-8$ edges, which is the upper bound of lemma 3.4, G is called *optimal* 1-planar [54] or *maximal* 1-planar [65].

The above proof shows an one-to-one correspondence between the family of optimal 1-planar graphs and the family of quadrangulations. Removing the crossing edges results in a quadrangulation, removing one edge of the two crossing edges, the resulting graph is a triangulation. This correlation was first discovered by Ringel, implied by Bodendiek et al., but formally proven by Fabrici and Madaras. In the work of Suzuki, it is further proven, that the quadrangulations must be 3-connected.

Now, we can give the following definition of optimal 1-planar graphs.

Definition 3.4. [12, p.324](*Optimal 1-planar graph*) Let Q be a simple quadrangulation of order $n > 2$ with $n-2$ faces and $2(n-2)$ edges. Adding the crossing diagonal edges in each of the $n-2$ faces, the resulting graph is called *optimal* 1-planar or *maximal* 1-planar, if no parallel edges occur.

Definition 3.5. [17, p.2](*Planar skeleton*) Given a 1-planar embedding G. Removing the crossing edges from G results in a simple quadrangulation Q. We call Q the *planar skeleton*, or simply *skeleton*, of G.

Example 3.6. Given a simple quadrangulation Q with $n = 8$ vertices. Q is optimal 1-planar if we add the crossing diagonal edges in each face, including the outerface of Q. The resulting graph G is shown in figure 3.4.

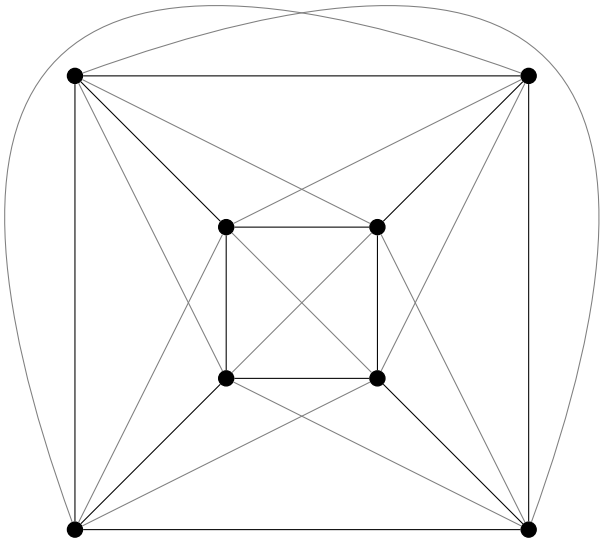


Figure 3.4: Optimal 1-planar graph with $n = 8$ vertices.

The graph G has $4 \cdot 8 - 8 = 24$ edges, satisfying the upper bound. The skeleton Q of G has $n - 2 = 8 - 2 = 6$ faces (including the outerface) and $2(n - 2) = 2 \cdot 6 = 12$ edges. The crossing diagonal edges are visualized in gray color in figure 3.4.

We are going to denote an optimal 1-planar graph of order k by $D_k^* = (V^*, E^*)$. Hence, the graph in example 3.6 is a D_8^*. D_k^* contains, by definition, a simple quadrangulation $Q = Q(D_k^*) = (V_Q^*, E_Q^*)$ of order n with $n-2$ faces and $2(n-2)$ edges. In order to generate D_k^* from $Q(D_k^*)$, we add 2 edges, the crossing diagonals, in each of the $n - 2$ faces.

The reason why the skeleton Q of an optimal 1-planar graph D^* with n vertices has exactly $2(n - 2)$ edges follows directly from the following theorem.

Theorem 3.1. Let $G = (V, E)$ be a connected, simple planar graph with $|V|$ vertices and $|E|$ edges. Denote C_k as a cycle of k vertices. If every face of G is isomorphic to C_k, then $|E| = \frac{k(|V|-2)}{k-2}$. [17]

[17] The theorem and the proof have both been adapted from [48].

Proof. Let F be the set of faces of G and let $s(f)$ be the number of edges in the face of f, then $2|E| = \sum_{f \in F} s(f)$, since every edge is adjacent to exactly two faces and every face is adjacent to $|f|$ edges. If each face is isomorphic to a cycle C_k, then

$$2|E| = \sum_{f \in F} k = k|F| \tag{67}$$

$$\Leftrightarrow \quad |F| = \frac{2|E|}{k} \tag{68}$$

$$\Rightarrow \quad 2 = |V| - |E| + \frac{2|E|}{k} \qquad \text{(Euler's formula)} \tag{69}$$

$$\Leftrightarrow \quad \frac{2-k}{k}|E| = 2 - |V| \tag{70}$$

$$\Leftrightarrow \quad |E| = \frac{k(|V|-2)}{k-2} \tag{71}$$

□

Using theorem 3.1 and definition 3.4 and setting $|V| = n$, it follows

$$|E| = \frac{4(n-2)}{4-2} = \frac{4(n-2)}{2} = 2(n-2) \tag{72}$$

Then, the upper bound suggested in lemma 3.4 follows directly from the definition:

$$|E^*| = 2|F(Q(D_k^*))| + 2(n-2) \tag{73}$$

$$= 2(n-2) + 2(n-2) \tag{74}$$

$$= 2(2n-4) \tag{75}$$

$$= 4n - 8 \tag{76}$$

Given a simple graph $G = (V, E)$, with $|V| = n$, then $d(v) \leq n-1,\ \forall v \in V$. Using the handshaking lemma, one can find the maximum number of edges possible.

$$\sum_{v \in V} d(v) = 2|E| \tag{77}$$

$$\Rightarrow n(n-1) = 2|E| \tag{78}$$

$$\Leftrightarrow \frac{n(n-1)}{2} = |E| \tag{79}$$

The above equation is especially the same as $\binom{n}{2}$.

$$\binom{n}{2} = \frac{n!}{2(n-2)!} = \frac{1 \cdot 2 \cdots (n-2) \cdot (n-1) \cdot n}{2(n-2)!} \tag{80}$$

$$= \frac{(n-2)!(n-1)n}{2(n-2)!} \tag{81}$$

$$= \frac{n(n-1)}{2} \tag{82}$$

Theorem 3.2. If a graph $G = (V, E)$ is optimal 1-planar, then $|V| \geq 7$.

Proof. From equation (79) it follows:

$$4n - 8 \leq \frac{n(n-1)}{2} \tag{83}$$

$$0 \leq n^2 - 9n + 16 \tag{84}$$

We solve the quadratic equation

$$n \geq \frac{9}{2} + \sqrt{\left(\frac{9}{2}\right)^2 - 16} \qquad n \leq \frac{9}{2} - \sqrt{\left(\frac{9}{2}\right)^2 - 16} \tag{85}$$

$$n \geq \frac{9}{2} + \sqrt{4.25} \approx 6.5615 \qquad n \leq \frac{9}{2} - \sqrt{4.25} \approx 2.4384 \tag{86}$$

The lower bound is excluded by definition. Since $n \in \mathbb{N}$, the upper bound is 7. □

Theorem 3.3. Given an optimal 1-planar graph $D^* = (V^*, E^*)$. The degree $d(v),\ v \in V^*$ is an even number [12].

Proof. There are $m \in \mathbb{N}$ faces incident to a vertex $v \in V^*$. Therefore, there are m diagonals in D^* incident to v (by definition). By definition, D^* contains a simple quadrangulation Q. Denote $d_Q(v)$ as the degree of v in Q and $d_{D^*}(v)$ as the degree of v in D^*. Clearly $d_{D^*}(v) = d_Q(v) + m$. If $d_Q(v)$ is even, then m is even. Hence, $d_{D^*}(v)$ is even (lemma A.3). If $d_Q(v)$ is odd, then m is odd. Hence, $d_{D^*}(v)$ is even (lemma A.2). □

In the following we are going to use a definition given by Joswig et al., in order to simplify the notation of the vertex-face incidence relationship.

Definition 3.7. [60, p.2](*vertex-face incidence matrix*) Given a planar graph G with m faces and n vertices. A 0/1-matrix $VF = (a_{fv}) \in \{0,1\}^{m \times n}$ is a *vertex-face incidence matrix* of G if the vertices and faces of G can be numbered by $\{0, \cdots, n\}$ and $\{0, \cdots, m\}$, respectively, such that $a_{fv} = 1$ if and only if the vertex with number v is incident to the face with number f.

Example 3.8. Given a simple quadrangulation $Q = (V, E)$ with $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$. The adjacency matrix is given by

$$A = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \\ v_{10} \end{array} \begin{array}{c} \begin{array}{ccccccccccc} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{87}$$

From A follows that $|E| = 18$. The number of faces of Q follows from Euler's formula: $|V| - |E| + |F| = 2 \Leftrightarrow |F| = |E| - |V| + 2 = 18 - 11 + 2 = 9$. The faces are given by

$$F_0 = v_0v_{10}v_9v_3v_0 \qquad F_1 = v_0v_1v_4v_{10}v_0 \qquad F_2 = v_1v_2v_7v_5v_1$$

$$F_3 = v_2v_3v_8v_7v_2 \qquad F_4 = v_3v_9v_6v_8v_3 \qquad F_5 = v_9v_{10}v_4v_6v_3$$

$$F_6 = v_8v_6v_5v_7v_8 \qquad F_7 = v_6v_4v_1v_5v_6 \qquad F_8 = \text{the outer face}$$

Notice that each face, except the outer face, is bounded by a 4-cycle, satisfying the definition. We are interested in the face-vertex incidence relationship.

We define the vertex-face incidence matrix by

$$VF_A = \begin{array}{c} \\ F_0 \\ F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \\ F_7 \\ F_8 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{88}$$

The degree of the vertices $v_{ij} \in V$ of Q are given by the column sums (or row sums) of A, where i is the row index and j is the column index of A. Therefore

$$d(v_0) = \sum_{i=0}^{10} v_{i0} = 3 \qquad d(v_1) = \sum_{i=0}^{10} v_{i1} = 4 \qquad d(v_2) = \sum_{i=0}^{10} v_{i2} = 3$$
$$d(v_3) = \sum_{i=0}^{10} v_{i3} = 4 \qquad d(v_4) = \sum_{i=0}^{10} v_{i4} = 3 \qquad d(v_5) = \sum_{i=0}^{10} v_{i5} = 3$$
$$d(v_6) = \sum_{i=0}^{10} v_{i6} = 4 \qquad d(v_7) = \sum_{i=0}^{10} v_{i7} = 3 \qquad d(v_8) = \sum_{i=0}^{10} v_{i8} = 3$$
$$d(v_9) = \sum_{i=0}^{10} v_{i9} = 3 \qquad d(v_{10}) = \sum_{i=0}^{10} v_{i10} = 3$$

In order to make Q optimal 1-planar, we add the crossing diagonals in each face of Q. Therefore, the degree of each vertex $v_{ij} \in V$ is increased by the number of faces incident with v_{ij}, where i is the row index and j is the column index of VF_A. We denote the number of faces of a vertex v by f_v. They are given by the column sums of VF_A

$$f_{v_0} = \sum_{i=0}^{8} v_{i0} = 3 \qquad f_{v_1} = \sum_{i=0}^{8} v_{i1} = 4 \qquad f_{v_2} = \sum_{i=0}^{8} v_{i2} = 3$$
$$f_{v_3} = \sum_{i=0}^{8} v_{i3} = 4 \qquad f_{v_4} = \sum_{i=0}^{8} v_{i4} = 3 \qquad f_{v_5} = \sum_{i=0}^{8} v_{i5} = 3$$
$$f_{v_6} = \sum_{i=0}^{8} v_{i6} = 4 \qquad f_{v_7} = \sum_{i=0}^{8} v_{i7} = 3 \qquad f_{v_8} = \sum_{i=0}^{8} v_{i8} = 3$$
$$f_{v_9} = \sum_{i=0}^{8} v_{i9} = 3 \qquad f_{v_{10}} = \sum_{i=0}^{8} v_{i10} = 3$$

The vertex degrees d_{D^*} of the optimal 1-planar graph $D^*(Q) = (V^*, E^*)$ with $V^* = V$ and $E^* = E \cup E_{fv}$, whereby E_{fv} are the crossing diagonals in each face of Q, are given by

$$d_{D^*}(v_0) = d(v_0) + f_{v_0} = 6 \qquad d_{D^*}(v_1) = d(v_1) + f_{v_1} = 8 \qquad d_{D^*}(v_2) = d(v_2) + f_{v_2} = 6$$
$$d_{D^*}(v_3) = d(v_3) + f_{v_3} = 8 \qquad d_{D^*}(v_4) = d(v_4) + f_{v_4} = 6 \qquad d_{D^*}(v_5) = d(v_5) + f_{v_5} = 6$$
$$d_{D^*}(v_6) = d(v_6) + f_{v_6} = 8 \qquad d_{D^*}(v_7) = d(v_7) + f_{v_7} = 6 \qquad d_{D^*}(v_8) = d(v_8) + f_{v_8} = 6$$
$$d_{D^*}(v_9) = d(v_9) + f_{v_9} = 6 \qquad d_{D^*}(v_{10}) = d(v_{10}) + f_{v_{10}} = 6$$

Notice that each degree is an even number. The simple quadrangulation Q is visualized in figure 3.5. Adding the crossing diagonals in each face results in the graph D^*, which is visualized in figure 3.6. D^* has excactly $4n - 8 = 4 \cdot 11 - 8 = 36$ edges, making it optimal 1-planar. The crossing diagonals in the faces of Q are colored in grey. We omit the labeling of the vertices for reasons of simplicity.

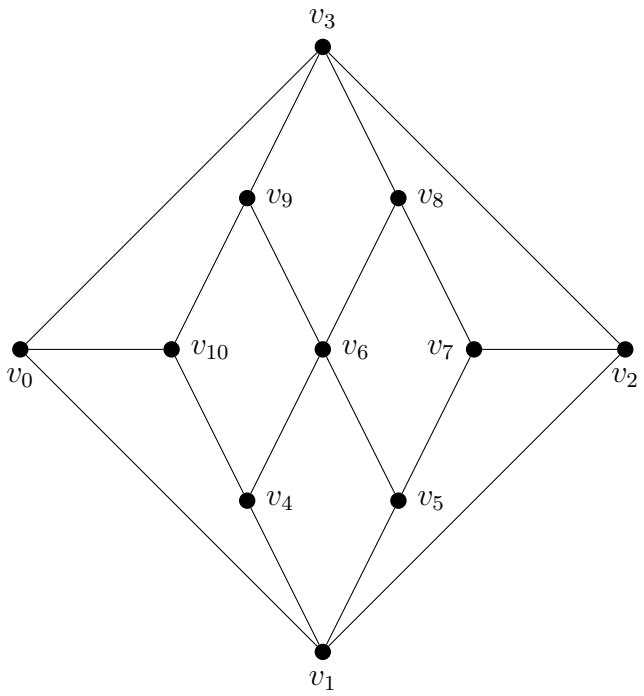


Figure 3.5: Simple quadrangulation Q with $n = 11$ vertices.

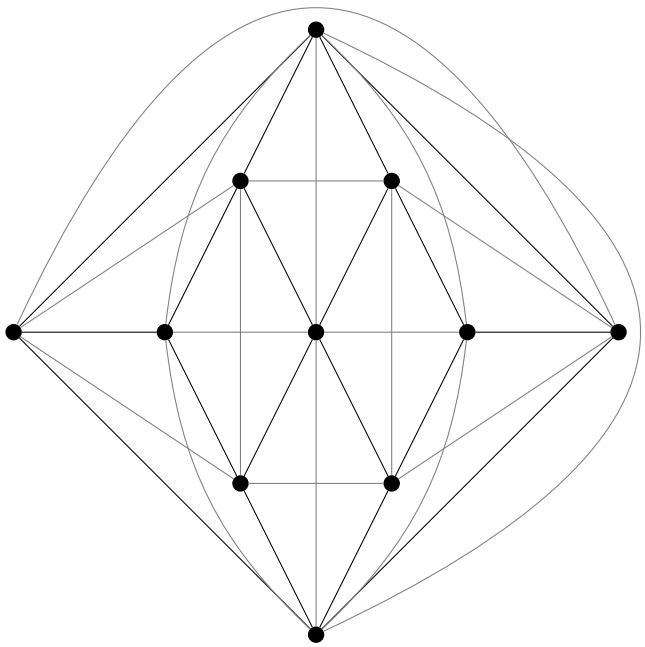


Figure 3.6: Optimal 1-planar graph with $n = 11$ vertices.

Using theorem 3.3, we can tighten theorem 3.2.

Theorem 3.4. There is no optimal 1-planar graph $D_7^* = (V^*, E^*)$ with $|V^*| = 7$.

Proof. Assume $D_7^* = (V^*, E^*)$ with $|V^*| = 7 = n$ is optimal 1-planar. Then, by definition, $|E^*| = 4 \cdot 7 - 8 = 20$. By equation (79) the maximum number of edges in a simple graph G with $n = 7$ vertices is given by $\frac{n(n-1)}{2} = 21$. Hence, $D_7^* = (V^*, E^*)$ must result after removing one edge of G. G is a complete graph, therefore each vertex of G has $n - 1 = 7 - 1 = 6$ neighbors, and therefore $d(v) = 6$, $\forall v \in V(G)$. But removing one edge from G results in a vertex with degree 5, contradicting theorem 3.3.[18] □

In section 3.1, we have seen that the skeleton $Q(D^*)$ of a given optimal 1-planar graph D^* is bipartite. Using this property and the properties, which we have studied in this section, one can derive the following theorem, which is proven in the work of Bodendiek et al.

Theorem 3.5. There is no optimal 1-planar graph $D_9^* = (V^*, E^*)$ with $|V^*| = 9$.

Additionally, the following theorem holds, as proven in [12].

Theorem 3.6. Let $D^* = (V, E)$ be an optimal 1-planar graph. For each vertex $v \in V$, it holds $d(v) \geq 6$.

Furthermore, it is noticeable that the crossing and non-crossing edges alternate in the rotation system of the embedding of an optimal 1-planar graph [17]. Additionally, it is important that the exact number of optimal 1-planar graphs is only known for graphs of size up to 36 [17]. This follows from the enumeration for simple quadrangulations given by Brinkmann et al., which is 3000183106119 for 36 vertices.

Summarizing this section, we have fore and foremost shown a relationship between the set of optimal 1-planar graphs and the set of simple quadrangulations. Especially, we have seen that optimal 1-planar graphs with n vertices only exist for $n = 8$ and $n \geq 10$, and that they do not exist for $n \leq 7$ and $n = 9$. We are going to use our investigations for the development of our generation algorithm. Furthermore, in [54] and [33] many interesting invariant relations and properties of optimal 1-planar graphs are presented.

[18] Cf. [12].

4 Generation of k-angulations

Before we examine a generation algorithm for simple quadrangulations, we are going to investigate the subject of graph generation at a higher level. We will first take a closer look at the generation of k-angulations, which are plane graphs in which the size of every face is k [59]. Then, we are going to investigate the generation of k-angulations with $k = 4$, which are exactly the simple quadrangulations as defined in the last sections. The results will be the basis for the generation of the optimal 1-planar graphs. We are especially interested in isomorph-free generation as we want exactly one representative of each equivalence class (under isomorphism) as an input for the automated graph conjecture-making.

4.1 Recursive generation

Definition 4.1. [59, p.8](*Recursive generation*) Assume $\mathcal{F}$ is a family of graphs and $\mathcal{U}$ is a superset of $\mathcal{F}, \mathcal{I} \subset \mathcal{U}$ and $\mathcal{E}$ a set of functions from $\mathcal{U}$ to $2^{\mathcal{U}-\mathcal{I}}$. Then $(\mathcal{I}, \mathcal{E}, \mathcal{U})$ is said to *recursively generate* $\mathcal{F}$, if for every graph $G \in \mathcal{F}$ there is a finite sequence $G_0, \cdots, G_n = G$ (n could be 0) such that $G_0 \in \mathcal{I}$ and $\forall i < n, G_{i+1} \in X(G_i)$ for some $X \in \mathcal{E}$. We call $(\mathcal{I}, \mathcal{E}, \mathcal{U})$ the *recursive generator* of $\mathcal{F}$.

Definition 4.2. [59, p.8](*Expansions*) Given a recursive generator $(\mathcal{I}, \mathcal{E}, \mathcal{U})$ for a family $\mathcal{F}$ of graphs. The members of $\mathcal{E}$ are called *expansions*.

Definition 4.3. [59, p.8](*Reductions*) Given a recursive generator $(\mathcal{I}, \mathcal{E}, \mathcal{U})$ for a family $\mathcal{F}$ of graphs, then for each expansion $X \in \mathcal{E}$ there is a function R_X that maps every graph $G \in \mathcal{F}$ to all graphs G' that can be expanded to G (the set can be empty) using X; mathematically speaking $R_X(G) = \{G' \in \mathcal{U} \ : \ G \in X(G')\}$. These functions are called *reductions*.

Definition 4.4. [59, p.8](*Irreducible graphs*) Given a recursive generator $(\mathcal{I}, \mathcal{E}, \mathcal{U})$ for a family $\mathcal{F}$ of graphs. Members of $\mathcal{I}$ and $\mathcal{U} - \mathcal{I}$ are called *irreducible graphs* and *reducible*, respectively.

Definition 4.5. [59, p.8](*Parents*) Given a recursive generator $(\mathcal{I}, \mathcal{E}, \mathcal{U})$ for a family $\mathcal{F}$ of graphs and the set of reductions R_X of $\mathcal{E}$. For a graph $G \in U$, all $G' \in R_X(G)$ for some $X \in \mathcal{E}$ are called *parents* of G.

Before we give an example for the above definitions, we introduce new conventions for visualizing graph embeddings as given by Brinkmann et al.

(i) Each displayed vertex is distinct from the others.

(ii) Edges that are completely drawn must occur in cyclic order given in the picture.

(iii) Half-edges indicate that an edge *must* occur at this position in the cyclic order around the vertex.

(iv) A triangle indicates that one or more edges *may* occur at this position in the cyclic order around the vertex (but they need not).

(v) If either a half-edge nor a triangle is present in the angle between two edges in the picture, then these two edges must follow each other directly in the cyclic ordering of edges around that vertex.

The term "half-edges" was originally introduced in [67] and used for the definition of combinatorial maps, where they are called *darts* or 1-*flags*.

Example 4.6. Given K_4, the complete graph with 4 vertices. The plane graph of K_4 is visualized in figure 4.1. K_4 consists of 3 triangular faces $T_0 = v_0v_2v_1v_0$, $T_1 = v_0v_2v_3v_0$ and $T_2 = v_1v_2v_3v_1$.

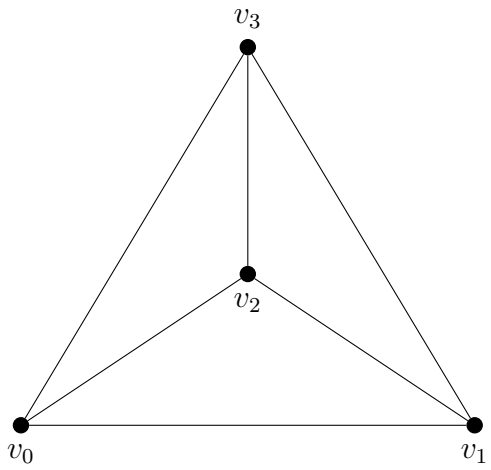


Figure 4.1: Complete graph K_4 with 4 vertices.

Using (i)-(v) we can visualize each triangle individually without loosing information about the structure in which they are embedded. We add an half-edge for each vertex, indicating that an edge must occur at this position (in cyclic order). The result is visualized in figure 4.2, we denote it by S_{K_4}.

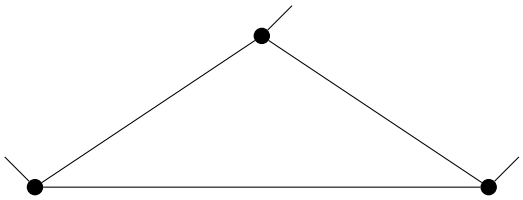


Figure 4.2: Substructure of K_4 visualized with half edges according to (iii).

Notice that in the interior no half-edge (or triangle) is present, therefore the edges follow each other directly in cyclic ordering around the vertex (compare with (v)). Now, we add a triangle before each half-edge of S_{K_4}. The result is visualized in figure 4.3, we denote it by S'_{K_4}. The minimum degree of each vertex of S'_{K_4} is 3.

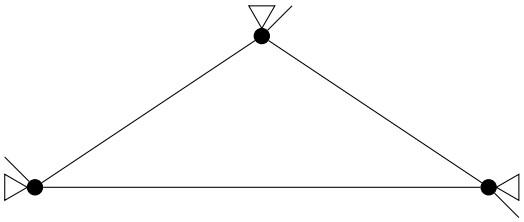


Figure 4.3: Graph with triangles and half edges according to (iii) and (iv).

One possible graph structure for S'_{K_4} is visualized in figure 4.4. We denote the graph by $G = (V, E)$ with $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ and $E = \{(v_0, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_0), (v_0, v_4), (v_4, v_1), (v_4, v_2), (v_4, v_5), (v_5, v_2), (v_5, v_1), (v_3, v_4)\}$.

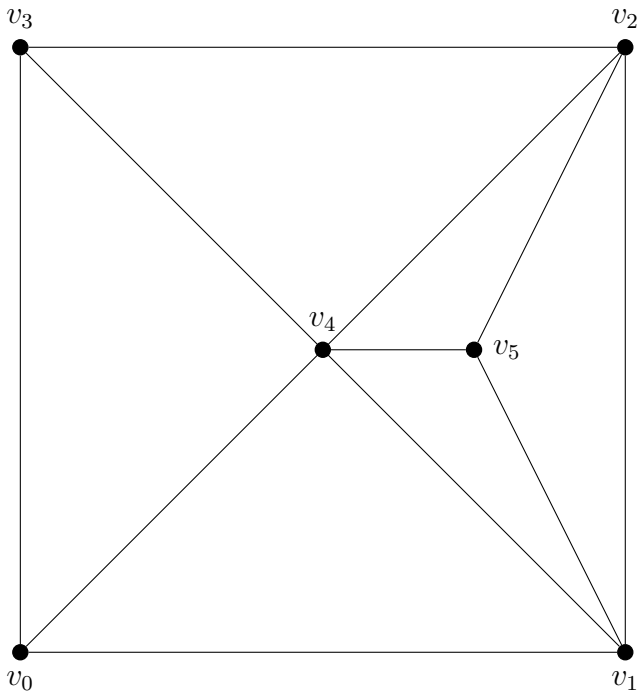


Figure 4.4: Possible graph structure of S'_{K_4}.

Let $G' = (V', E')$ with $V' = \{v_1, v_4, v_5\}$ and $E' = \{(v_1, v_4), (v_1, v_5), (v_4, v_5)\}$ be a subgraph of G. If we want to indicate the structure of G' in G, we could use S'_{K_4}. The edges (v_4, v_0) and (v_4, v_3) become a triangle. The edge (v_4, v_2) become an half-edge. Notice the correct circular ordering of the edges and how the triangle indicates multiple edges. The edge (v_5, v_2) becomes an half-edge. The edge (v_1, v_2) becomes a triangle. The edge (v_1, v_0) becomes an half-edge.

Next, we are going to generate the family of simple plane triangulations of order at least 4 to give an example for definitions 4.1,4.2, 4.3, 4.4 and 4.5. The example is taken from [21] and adopted to match our definitions. The generation algorithm is implemented in the tool *plantri*.

Example 4.7. Let $\mathcal{F}$ be the family of simple plane triangulations of order at least 4. Then, according to definition 4.1, $\mathcal{U} \supset \mathcal{F}$. It is convenient to set $\mathcal{F} = \mathcal{U}$, "which means the generator starts with some irreducible graphs from $[\mathcal{F}]$ and expands them to larger ones, while remaining inside $[\mathcal{F}]$" [59, p.9]. Following this convenience, we set $\mathcal{F} = \mathcal{U}$. We set $\mathcal{I} \subset \mathcal{F}$. By definition 4.4, the members of $\mathcal{I}$ are irreducible, hence for each $i \in \mathcal{I}$ there is no reduction to a graph $G \in \mathcal{F}$. Obviously $K_4 \in \mathcal{F}$ is irreducible in $\mathcal{F}$, since $\mathcal{F}$ contains all simple plane triangulations of order at least 4.
We define three rules $E_3, E_4, E_5 \in \mathcal{E}$. The expansion rules are defined as such (cf. [21, p.4])

1. E_3: A vertex v_x is added in the face interior of a triangulation with vertex set $\{v_0, v_1, v_2\}$. Then, the edges $(v_x, v_0), (v_x, v_1), (v_x, v_2)$ are added. The result is a triangulation with 6 edges and 4 vertices and 3 triangulations as faces.

2. E_4: If there is a subgraph with vertex set $\{v_0, v_1, v_2, v_3\}$ and $C_4 = v_0v_1v_2v_3v_0$ is a 4-cycle. Furthermore, there is an edge (v_0, v_2), the diagonal of the 4-cycle. It follows, that the graph has two triangular faces. E_4 deletes the diagonal edge, then adds a vertex v_x in the resulting square, and finally adds the edges $(v_x, v_0), (v_x, v_1), (v_x, v_2)$, (v_x, v_3). The resulting graph has 5 vertices, 8 edges and 4 triangular faces.

3. E_5: Given a graph G with vertex set $\{v_0, v_1, v_2, v_3, v_4\}$ and $C_5 = v_0v_1v_2v_3v_4v_0$ form a cycle. Furthermore, $e_x = (v_0, v_3), e_y = (v_1, v_3)$ are in the edge set of G. Each face is a triangulation. E_5 deletes e_x, e_y and adds a vertex v_x in the interior of the resulting pentagon. Then, the edges $(v_x, v_0), (v_x, v_1), (v_x, v_2), (v_x, v_3), (v_x, v_4)$ are added. The resulting graph has 5 triangular faces.

The above expansions were originally discovered by Berhard and implemented in the tool *plantri* by Brinkmann and McKay. E_3, E_4, E_5 are visualized in figure 4.5. The starting graph for the recursive generation is $K_4 \in \mathcal{I} \subset \mathcal{F}$.

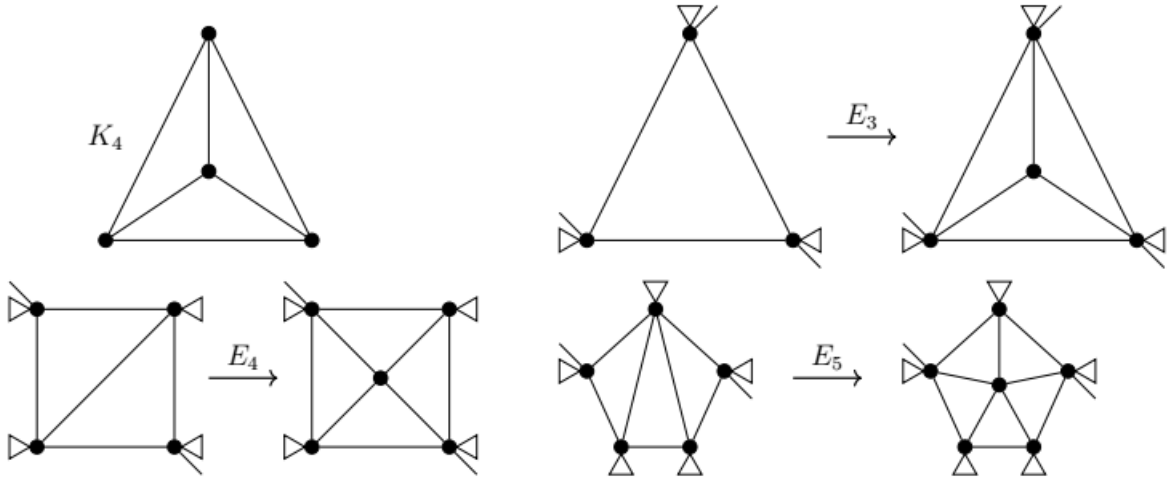


Figure 4.5: Expansion rules E_3, E_4, E_5 for generating simple triangulations.

In this section we have introduced the topic of recursive graph generation by giving the necessary definitions. Furthermore, we introduced a few new rules for visualizing substructures of graph embeddings as given by Brinkmann et al. in order to simplify the process of visualizing the expansion rules. Finally, we have given an example for generating simple triangulations.

4.2 Generation of simple quadrangulations

In the following section we are going to investigate an algorithm for generating simple quadrangulations. The algorithm starts with a set of irreducible graphs of the given graph family and then performs a sequence of expansions on them, which expands them to larger graphs while staying in the same graph family. The algorithm was introduced by Brinkmann et al. We are going to present the algorithm by investigating the set of expansion rules and by giving a few examples.

Following the definition given in section 4.1, the generator for simple quadrangulations is defined as $(\mathcal{I}, \mathcal{F}, \mathcal{E})$. We denote the set of simple quadrangulations by $\mathcal{F}$. The starting set $\mathcal{I}$ will either consists of the square or the pseudo-double wheels [22].

Definition 4.8. [22, p.34](*Pseudo-double wheel*) Given a simple quadrangulation Q with $n \geq 8$ vertices, whereby n is even. Q consists of a cycle $C = v_0 v_1 \cdots v_{n-3}$, as well as a vertex which is adjacent to $v_0, v_2, \cdots, v_{n-4}$ and a vertex which is adjacent to $v_1, v_3, \cdots, v_{n-3}$. We call Q a *pseudo-double wheel*. The smallest double-wheel is called a *cube*. We are going to denote a pseudo-double wheel by W_n, whereby n is the number of vertices.

Example 4.9. Given a graph $W_{10} = (V, E)$ with $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$ and $E = \{(v_0, v_1), (v_0, v_9), (v_0, v_7), (v_1, v_2), (v_1, v_8), (v_2, v_9), (v_2, v_3), (v_3, v_4), (v_3, v_8), (v_4, v_5), (v_4, v_9), (v_5, v_8), (v_5, v_6), (v_6, v_7), (v_6, v_9), (v_7, v_8)\}$. The graph is visualized in figure 4.6.[19]

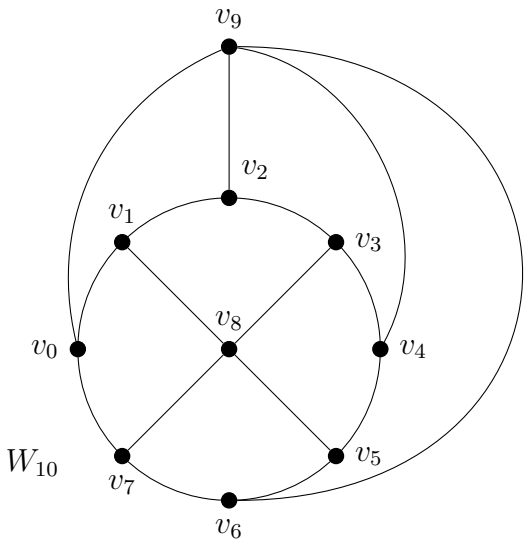


Figure 4.6: Pseudo-double wheel with $n = 10$ vertices.

$|V| = 10 = n$, thus W_{10} has more than 8 vertices, and n is even. Furthermore, $C = v_0v_1v_2v_3v_4v_5v_6v_7v_0$ is a cycle ($v_{n-3} = v_7$). The vertex v_8 is adjacent to v_1, v_3, v_5, v_7. The vertex v_9 is adjacent to (v_0, v_2, v_4v_6). W_{10} satisfies definition 4.8, thus W_{10} is a pseudo-double wheel.
Given a graph $W_8 = (V', E')$ with $V' = \{v'_0, v'_1, v'_2, v'_3, v'_4, v'_5, v'_6, v'_7\}$ and $E' = \{(v'_0, v'_1), (v'_0, v'_5), (v'_0, v'_7), (v'_1, v'_6), (v'_1, v'_2), (v'_2, v'_3), (v'_2, v'_7), (v'_3, v'_4), (v'_3, v'_6), (v'_4, v'_7), (v'_4, v'_5), (v'_5, v'_6)\}$ has $n = 8$ vertices. The number of vertices is even. Furthermore, $C' = v'_0v'_1v'_2v'_3v'_4v'_5v'_0$ is a cycle ($v'_{n-3} = v'_5$).

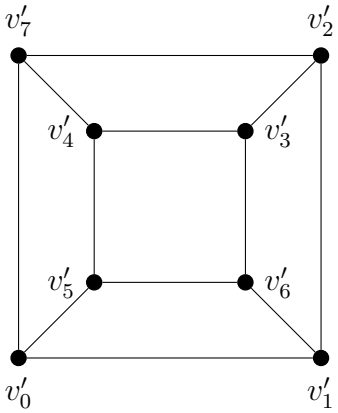


Figure 4.7: Smallest pseudo-double wheel W_8 with $n = 8$ vertices.

[19]The shown embedding of W_{10} differs from the one given by [22]. However, it coincides with the embedding given by Suzuki. In example 2.51 we have shown that the two graphs are isomorphic. We have choosen this embedding, because the cycle, demanded by the definition, is immediately recognizable.

The vertex v_6' is adjacent to v_1', v_3', v_5'. The vertex v_7' is adjacent to v_0', v_2', v_4'. Therefore, W_8 is a pseudo-double wheel. W_8 is visualized in figure 4.7.

We define the expansions in $\mathcal{E}$ as the inverse of the corresponding reduction [22, p.35]. More specifically, we call the reduction a *face contraction*.

Definition 4.10. [22, p.35](*Face contraction*) Let Q be a simple quadrangulation with a face $F = (x, u, v, w)$. The *contraction* of F at the vertices x, v produces a quadrangulation Q' obtained from Q by identifying the vertices x and v to form a new vertex x', identifying the edges (x, u) and (v, u) to form a new edge (x', u), and identifying the edges (x, w) and (v, w) to form a new edge (x', w). The faces of Q' are the faces of Q other than F.

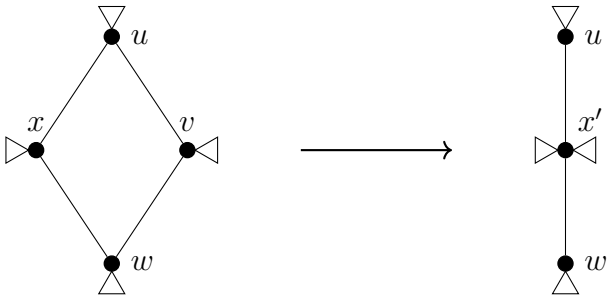


Figure 4.8: Example of a face contraction.

We continue with the definition of the reduction rules.

Definition 4.11. [22, p.35-36](P_0-*reduction*) Suppose there are distinct faces (u, v, w, x) and (u, y, w, v), so that v has degree 2. A P_0-*reduction* consists of a face contraction at $\{x, v\}$.

Example 4.12. Given a face $F_0 = (y, u, x', w)$ (compare with left side of figure 4.9). A P_0-expansion adds a vertex in the interior of F_0 and adds the edges (w, v) and (u, v). Thus, the faces (y, u, v, w) and (u, x, w, v) are created. Similarly, a P_0-reduction merges the vertices x, v to a single vertex x'. Thus, the face (y, u, x', w) is created. The P_0-expansion is visualized in figure 4.9.

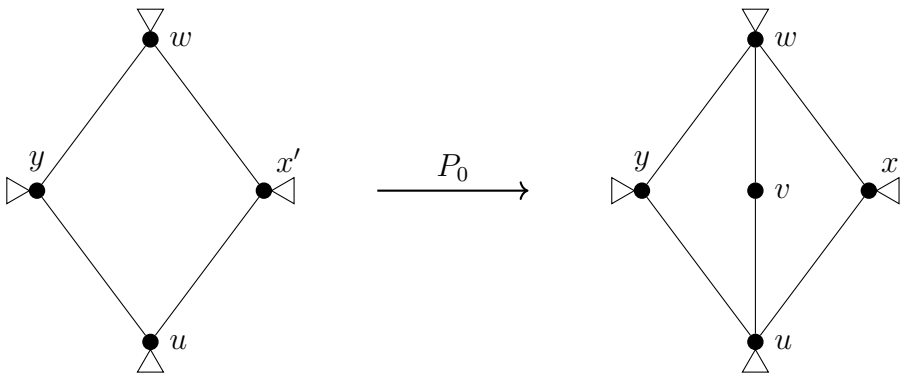


Figure 4.9: P_0-expansion.

Definition 4.13. [22, p.35-36](P_1-*reduction*) A P_1-*reduction* consists of a contraction of a face (x, u, v, w) at $\{x, v\}$, where x has degree 3 and u, v and w each have degree at least 3.

Example 4.14. In figure 4.10 a P_1-expansion is visualized. The vertex x' is splitted into two vertices v, x. The edges (w, v),(w, x),(v, u),(x, u) are added. Thus, the face (x, u, v, w) is created. Notice, that x is incident to two full edges and one half edge, and therefore has degree of excactly 3. The vertices u, v, w are incident to 3 edges at minimum.

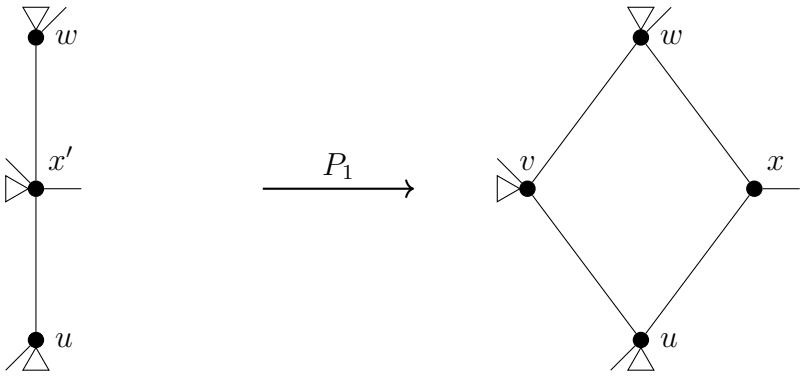


Figure 4.10: P_1-expansion.

The triangle, placed at each vertex u, v, w, indicates that one or more additional edges could occur in cyclic order around each vertex. Therefore, the degree of u, v and w is at least 3.

Definition 4.15. [22, p.35-36](P_2-*reduction*) A P_2-*reduction* is defined as a sequence of two face contractions. We have faces labelled clockwise as follows: (p, t, y, u),(t, z, x, y), (u, y, x, w),(z, v, w, x),(t, q, v, z). All labelled vertices except t and possibly w have degree excactly 3, while t has degree at least 4 and w has degree at least 3. A P_2-*reduction* consists of a face contraction at $\{x, v\}$ followed by a face contraction at $\{z, q\}$.

Example 4.16. In figure 4.11 a P_2-expansion is visualized. The vertex x' is splitted into two vertices x and v. The vertex q' is splitted into the vertices q and z. The edges $(w, v), (v, z), (x, z), (y, z), (v, q)$ and (q, t) are added. The expanded graph has the faces $(p, t, y, u), (t, z, x, y), (u, y, x, w), (z, v, w, x)$ and (t, q, v, z).
We are going to graphically show the two face contractions of P_2 in order to give a better idea of how P_2 works. The vertices $\{v, x\}$ and $\{z, q\}$ are merged to a single vertex x' and q', respectively. In figure 4.12 the contraction at $\{x, v\}$ to x' is shown. Notice how the edges (v, q) and (w, v) become (x', q) and (x', w), respectively.

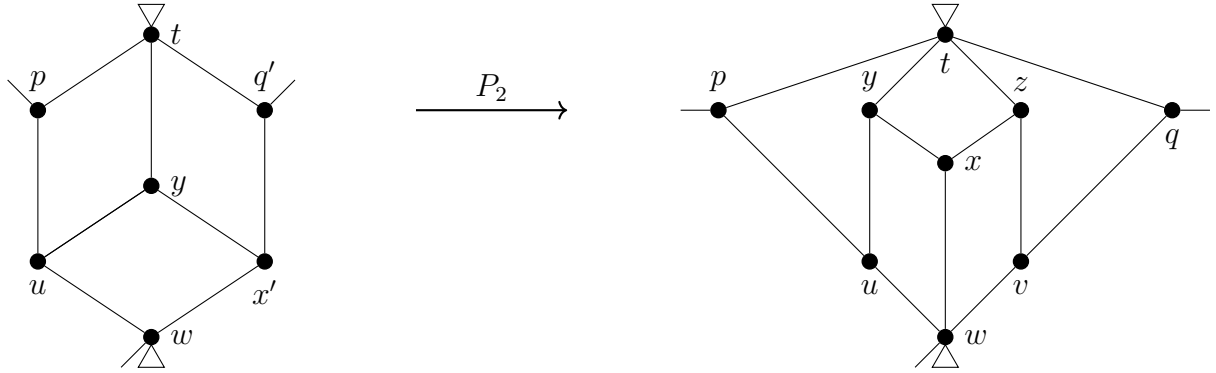


Figure 4.11: P_2-expansion.

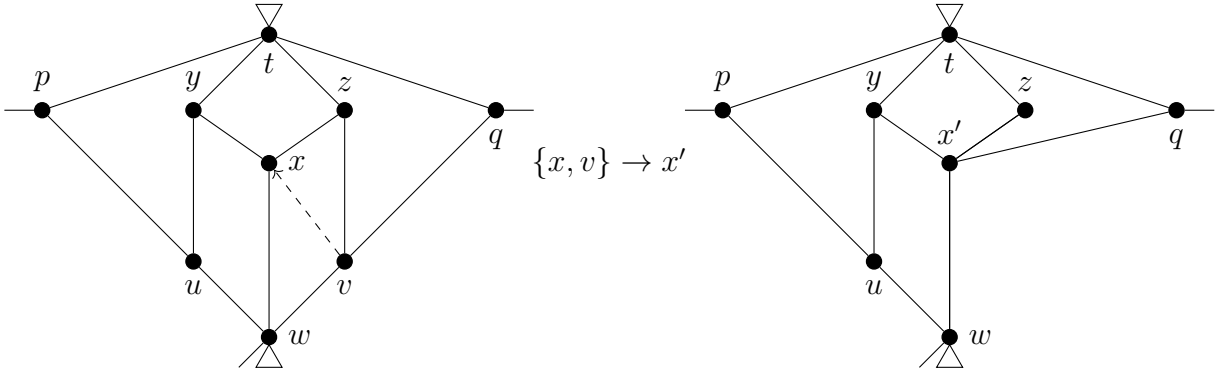


Figure 4.12: Face contraction at $\{x, v\}$.

The face (t, z, v, q) becomes (t, z, x', q) and (t, y, x, z) becomes (t, y, x', z). The face (x, w, v, z) is contracted, thus the resulting quadrangulation has one face less then the original one. In figure 4.13 the contraction at $\{z, q\}$ to the single vertex q' is shown. Notice how the edges (x', q) and (q, t) become (x', q') and (q', t), respectively. The face (t, z, x', y) becomes (t, q', x', y'). The face (t, q, x', z) is contracted, thus the resulting quadrangulation has one face less then the one after the $\{v, x\}$-contraction.

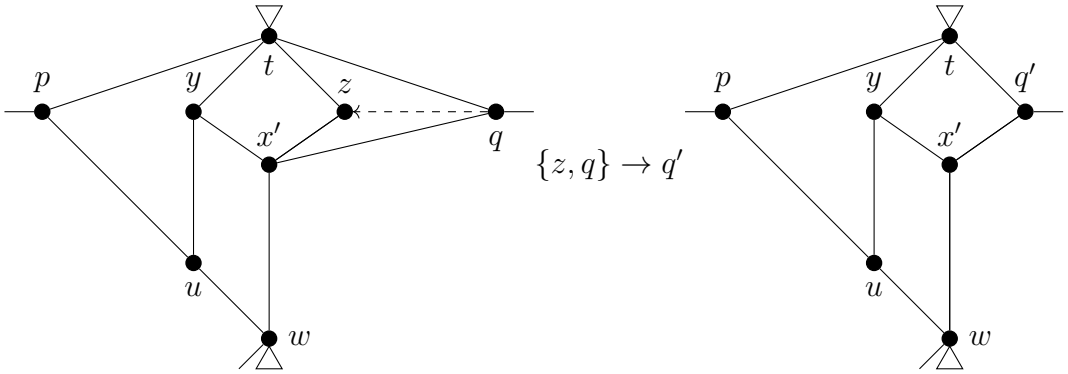


Figure 4.13: Face contraction at $\{z, q\}$.

Rearranging the vertices results in the quadrangulation before the P_2-expansion was applied (compare with figure 4.14).

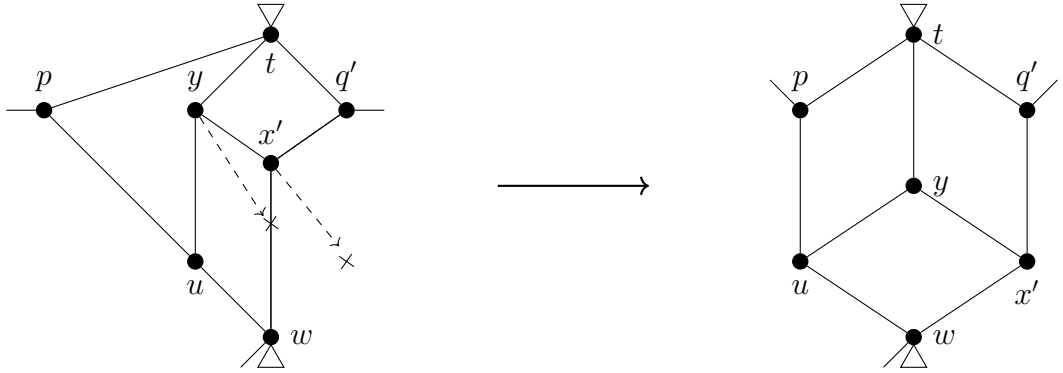


Figure 4.14: Rearranging the vertices.

Definition 4.17. [22, p.35-36](P_3-*reduction*) A P_3-*reduction* is defined as a sequence of four contractions. We have faces (u, v, w, x),(a, b, v, u),(b, c, w, v),(c, d, x, w) and (d, a, u, x). Here u, v, w and x all have degree 3, and we assume that a, b, c, d all have a degree of at least 4. A P_3-*reduction* consists of a face contraction at $\{a, v\}$, followed by one at $\{b, w\}$, followed by one at $\{c, x\}$, followed by one at $\{d, u\}$. Thus, a P_3-*reduction* is essentially just the removal of the vertices u, v, w, x and their incident edges.

Example 4.18. The P_3-expansion is visualized in figure 4.15. The vertex a' is splitted into the vertices a and v. The vertex b' is splitted into the vertices b and w. The vertex c' is splitted into the vertices c and x. The vertex d' is splitted into the vertices d and u. The edges (a, u), (b, v), (c, w) and (d, x) are added. The expanded graph has 4 more faces than the original one.

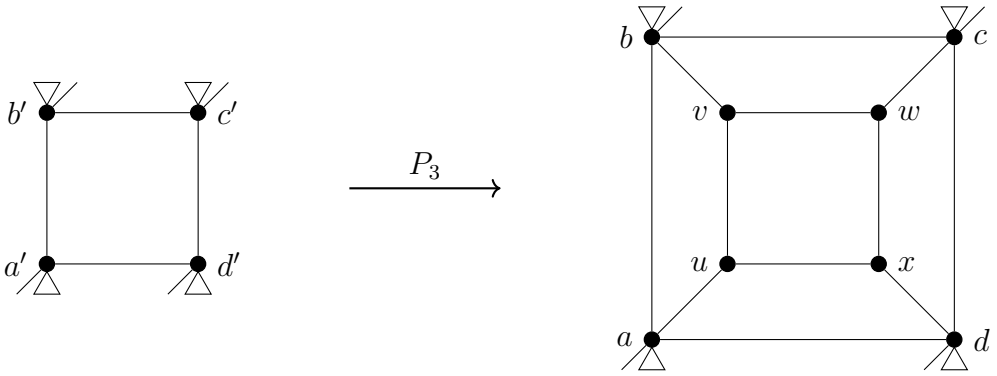


Figure 4.15: P_3-expansion.

Definition 4.19. [22, p.35-36](P_4-*reduction*) A P_4-*reduction* consists of a contraction of the face (x, u, v, w) at $\{x, v\}$, whereby each of x, u, v, w has degree at least 2.

Example 4.20. The P_4-expansion is visualized in figure 4.16. The vertex x' is splitted into the vertices x and v. The resulting face is (u, v, w, x). Notice the similarity with the P_1-expansion. In contrast to P_1 the vertices x, u, v, w have at least a degree of 2, while for P_1 the vertex x has a degree of 3 and u, v, w a degree of at least 3.

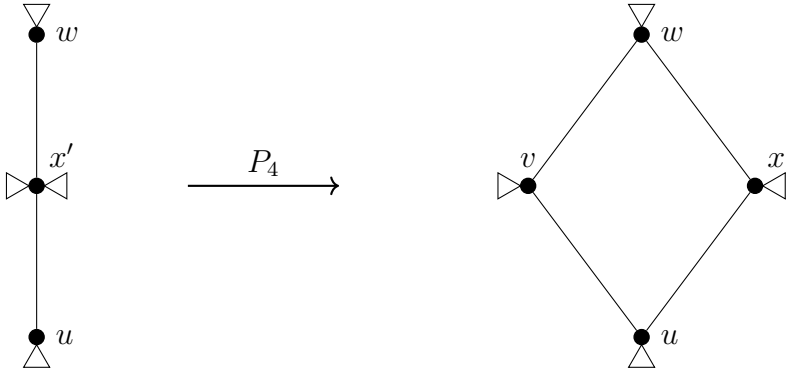


Figure 4.16: P_4-expansion.

Now, using the expansion rules, the generation theorems are as follows:

Theorem 4.1. [22, p.36] The class of all simple quadrangulations of the sphere is generated from the square by the P_0-expansions and P_1-expansions.

Theorem 4.2. [22, p.36] The class of all simple quadrangulations of the sphere with minimum degree 3 is generated from the pseudo-double wheels by the P_1-expansions and P_3-expansions.

Theorem 4.3. [22, p.36] The class of all 3-connected quadrangulations of the sphere is generated from the pseudo-double wheels by the P_1-expansions and P_3-expansions.

Theorem 4.4. [22, p.36] The class of all 3-connected quadrangulations of the sphere without separating 4-cycles is generated from the pseudo-double wheels by the P_1-expansions.

The proofs of the above theorems will not be repeated at this point. They can be found in [22].

The generation algorithm is implemented in the tool *plantri*, which is written in the *C* programming language. The program is open source and regularly updated. In the last sections we have seen that optimal 1-planar graphs can be constructed by adding the crossing diagonals in the faces of the simple quadrangulations. Therefore, we are going to process the output of plantri in order to generate optimal 1-planar graphs. To do so, we first need to study some characteristics of plantri, which are important for our further investigations. First and foremost, "isomorphisms are defined with respect to the embedding" [19], meaning the definition differs from the one given for abstract graphs. This is particularly important because "in some cases the ouputs may be isomorphic to abstract graphs" [19]. Secondly, plantri offers different output formats for the generated graphs. Since we want to use the generated output as an input for an automated conjecture-making program, we need to examine the compatibility of the output format.

4.2.1 O-P, O-R isomorphism

We start with the definition of graph isomorphism, implemented in plantri.

Definition 4.21. [21](*O-P* isomorphism) Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. An Orientation-preserving (O-P) isomorphism from G to G' is a bijection $\varphi : V \to V'$ and a bijection $\psi : E \to E'$, such that

(i) $(x, y) \in E \iff (\varphi(x), \varphi(y)) \in E' \ \forall x, y \in V$.

(ii) If $\{e_0, e_1, \cdots, e_k\}$ is the cyclic order of the edges of G incident with $v \in V$, then $\{\psi(e_0), \psi(e_1), \cdots, \psi(e_k)\}$ is the cyclic order of the edges incident with $\varphi(v) \in V'$.

Notice, that (i) is identical with the definition of abstract graph isomorphism, which we have given in section 2.2. (ii) demands that an isomorphism from a graph G to a graph G' respects the actual embedding.

Example 4.22. Given a graph $G = (V, E)$ with $V = \{v_0, v_1, v_2, v_3\}$ and $E = \{e_0 = (v_0, v_2), e_1 = (v_2, v_3), e_2 = (v_3, v_4), e_3 = (v_0, v_4), e_4 = (v_0, v_1)\}$. Furthermore, assume the rotation system of $\Pi_G = \{\pi_{v_0}, \pi_{v_1}, \pi_{v_2}, \pi_{v_3}, \pi_{v_4}\}$ of G, in clockwise order, is defined by

$$\pi_{v_0} = \begin{pmatrix} e_0 & e_3 & e_4 \\ e_3 & e_4 & e_0 \end{pmatrix} \qquad \pi_{v_1} = \begin{pmatrix} e_4 \\ e_4 \end{pmatrix} \qquad \pi_{v_2} = \begin{pmatrix} e_0 & e_1 \\ e_1 & e_0 \end{pmatrix} \tag{89}$$

$$\pi_{v_3} = \begin{pmatrix} e_1 & e_2 \\ e_2 & e_1 \end{pmatrix} \qquad \pi_{v_4} = \begin{pmatrix} e_2 & e_3 \\ e_3 & e_2 \end{pmatrix} \tag{90}$$

Given an (abstract) automorphism $\varphi : V \to V : v \mapsto v, \forall v \in V$ from G to itself. φ is the identity isomorphism of G. By definition of φ, (i) of definition 4.21 is satisfied. Define $\psi : E \to E : e \mapsto e, \forall e \in E$. Hence, for the cyclic order $\{e_x, \cdots, e_y\}$ around a vertex $v \in V$ holds $\{e_x, \cdots, e_y\} = \{\psi(e_x), \cdots, \psi(e_y)\}$. Obviously, (φ, ψ) defines an O-P isomorphism, or more specific an O-P automorphism.
Assume, we have the same (abstract) automorphism, but a different rotation system. Assume, that instead of $\pi_{v_0} = \begin{pmatrix} e_0 & e_3 & e_4 \\ e_3 & e_4 & e_0 \end{pmatrix}$ we have $\begin{pmatrix} e_0 & e_3 & e_4 \\ e_4 & e_0 & e_3 \end{pmatrix}$, then (ϕ, ψ) would still be an isomorphism, but not an O-P isomorphism, since the embedding is different. The two embeddings are visualized in figure 4.17. If one consider the actual embedding of the graphs for the graph isomorphism problem, then we also speak of *topological isomorphism* [32].
To give a conclusion: implementing O-P isomorphism in a graph generation tool, in order to reject isomorphic graphs, guarantees embedded-non-isomorphism but does not guarantee abstract non-isomorphism.

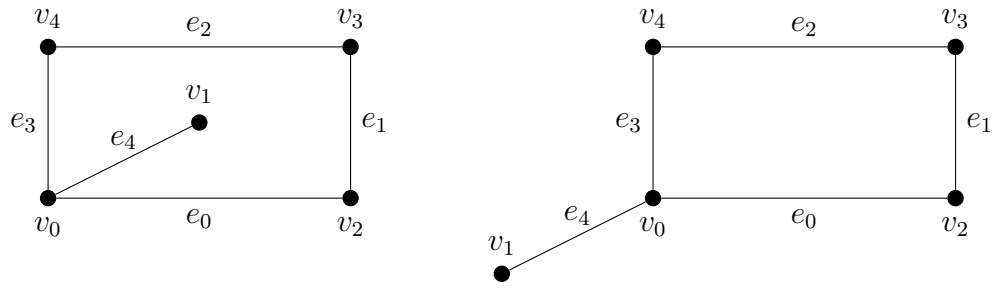


Figure 4.17: Two graphs being isomorphic, but not O-P isomorphic.

Remembering the results from section 2.2, the following theorem can be conjectured.

Theorem 4.5. O-P isomorphism is an equivalence relation.

In the following, we denote O-P isomorphism by $\hookrightarrow$.

Proof. Reflexivity, symmetry, and transitivity of (i) of definition 4.21 is proven in section 2.2. We prove (ii) of definition 4.21.

(i) *Reflexivity*: Define an abstract isomorphism from a graph $G = (V, E)$ to itself by $\varphi : V \to V : v \mapsto v$ and $\psi : E \to E : e \mapsto e$. Let $\{e_0, \cdots, e_k\}$ be the cyclic order of edges of G incident with $v \in V$. It holds $\{\psi(e_0), \cdots, \psi(e_k)\} = \{e_0, \cdots, e_k\}$, hence the cyclic order is preserved. Therefore, $G \hookrightarrow G$.

(ii) *Symmetry*: Given two graph embeddings $G = (V, E)$ and $G' = (V', E')$. Let $G \hookrightarrow G'$, then $\psi : E \to E', \varphi : V \to V'$ preserves the rotation system. Hence, if $\{e_0, \cdots, e_k\}$ is the cyclic order of the edges of G incident with $v \in V$, then by definition $\{\psi(e_0), \cdots, \psi(e_k)\}$ is the cyclic order of the edges of G' incident with $\varphi(v) \in V'$. Since φ and ψ are bijections, their inverses $\varphi^{-1} : V' \to V$ and $\psi^{-1} : E' \to E$ exist. We can conclude $\{\psi^{-1}(e_0), \cdots, \psi^{-1}(e_k)\} = \{e_0, \cdots, e_k\}$. Hence, $G' \hookrightarrow G$. And therefore, $G \hookrightarrow G' \Rightarrow G' \hookrightarrow G$.

(iii) *Transitivity*: Given three graph embeddings $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ and $G_3 = (V_3, E_3)$. Assume, that $G_1 \hookrightarrow G_2$ and $G_2 \hookrightarrow G_3$. Then, by definition, we have the bijections $\varphi_{12} : V_1 \to V_2, \psi_{12} : E_1 \to E_2$ and $\varphi_{23} : V_2 \to V_3, \psi_{23} : E_2 \to E_3$, which preserves the rotation system. Define $\varphi_{13} : V_1 \to V_2 : \varphi_{23} \circ \varphi_{12}$ and $\psi_{13} : E_1 \to E_3 : \psi_{23} \circ \psi_{12}$. Since $\varphi_{12}, \varphi_{23}, \psi_{12}, \psi_{23}$ are bijective, their composits are also bijective. Now, if $\{e_0, \cdots, e_k\}$ is the cyclic order of the edges of G_1 around $v \in V_1$, then $\{\psi_{12}(e_0), \cdots, \psi_{12}(e_k)\}$ is the cyclic order of the edges of G_2 around $\varphi_{12}(v) \in V_2$, and $\{\psi_{23}(\psi_{12}(e_0)), \cdots, \psi_{23}(\psi_{12}(e_k))\}$ is the cyclic order of the edges of G_3 around $\varphi_{23} \in V_3$, which is the same as $\{\psi_{13}(e_0), \cdots, \psi_{13}(e_k)\}$ for $\varphi_{13} \in V_3$. Therefore, $G_1 \hookrightarrow G_2 \wedge G_2 \hookrightarrow G_3 \Rightarrow G_1 \hookrightarrow G_3$.

□

Since O-P isomorphism is an equivalence relation, we can divide a set of graphs into *O-P isomorphism classes* just like we can do with abstract graph isomorphism (recap section 2.2).

Definition 4.23. [21](*O-R* isomorphism) Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. An Orientation-reversing (O-R) isomorphism from G to G' is a bijection $\varphi : V \to V'$ and a bijection $\psi : E \to E'$, such that

(i) $(x, y) \in E \iff (\varphi(x), \varphi(y)) \in E' \ \forall x, y \in V$.

(ii) If $\{e_0, e_1, \cdots, e_k\}$ is the cyclic order of the edges of G incident with $v \in V$, then $\{\psi(e_k), \psi(e_{k-1}), \cdots, \psi(e_0)\}$ is the cyclic order of the edges incident with $\varphi(v) \in V'$.

Notice, that definition 4.23 differs from definition 4.21 in that it reverses the cyclic order at each vertex [21].

Theorem 4.6. O-R isomorphism is not an equivalence relation.

The above theorem can be proven using the symmetry property of an equivalence relation. We omit a formal proof at this point.

4.2.2 The generation tool plantri

In the following, we are going to give a short introduction to the generation tool *plantri* for planar graphs as we will use it as a basis for the generation of optimal 1-planar graphs. The tool was implemented by Gunnar Brinkmann from the University of Ghent and Brendan McKay from the Australian National University [20]. It is regularly updated, and we are going to use the newest version, which is version 5.0, released 2018. [P]lantri is open source and written in the *C* programming language. The generation speed is very fast. For some graph classes it reaches $1,000,000$ graphs per second [20]. We are interested in generating simple quadrangulations, as they are the skeletons of optimal 1-planar graphs. A generation speed up to $400,000$ quadrangulations per second can be reached [22].

In the following, we are first going to give a quick guide on how to install and on how to use basic functions of plantri. Second, we are going to give an overview of the different output formats. They are particularly important as we want to process the generated graphs to an automated conjecture-making tool. Furthermore, we are going to present the usage of plantri in the software *SageMath* (or simply *sage*), since one can integrate plantri in sage and it implements many useful functions for graph theory.

Unless otherwise stated, the following commands are taken from [19] and tested on a linux machine (Linux Ubuntu 18.04 LTS). In order to compile plantri, we can use the Gnu Compiler Collection (GCC), which is a widely known collection of compilers for many different programming languages. The procedure for installing the GCC is shown in listing 1.

```
1 sudo apt-get update #update
2 sudo apt-get install build-essential #install gcc
3 gcc --version #validate successful installation
```

Listing 1: Installing gcc on ubuntu

First, we download the package lists from the package repositories and update the information for the newest versions of all packages and their dependencies (l.1). Then, we install the GCC (l.2). Finally, we validate the installation by checking the installed version (l.3).
We navigate to the plantri directory and compile the program using the *C* compiler of the GCC. The commands are shown in listing 2.

```
1 cc -o plantri -O4 plantri.c
2 chmod +x plantri #might be necessary
```

Listing 2: Compiling plantri

It might be necessary to grant executable permissions, depending on the machine (l.2). Generating all simple quadrangulations is rather simple. For example, generating all 3-connected planar quadrangulations with 8 vertices and printing them to standard output (stdout) is done with the command shown in listing 3.

```
1 plantri -q 8
```

Listing 3: Generating 3-connected planar quadrangulations

There is exactly one simple quadrangulation with 8 vertices up to isomorphism, which is proven in [12]. We have shown one possible embedding in section 3.1. In order to send the output of plantri to a file, one can specifiy a file name. There are numerous different options and different flags implemented in plantri. A complete overview can be found in [19].

In the following we are going to give an overview of the different output formats of plantri and give a small example for planar code as the main output format of plantri.

1. *Planar code*: This is the standard output format of plantri. It is a binary format, whereby each entry is an unsigned char. The vertices are numbered starting with 1. The first entry of the stream is equal to the number of vertices n. Then, there are n sections, starting with 1. Each section represents a vertex of the graph, and contains its neighbors in clockwise order, e.g. the first section contains the neighbors of vertex 1 in clockwise order. Each section ends with a 0. A file, containing planar code, begins with >>*planar_code*<<, whereby no end-of-line is appended. Planar code can be translated to an adjacency list using the open source program *planarread* by Brinkmann et al.

2. *Graph*6: This is a very space-efficient format, intended for small graphs [74]. It stores graph data by only using printable ASCII-characters. It was invented by McKay. The formal definition can be found in [74]. The program *showg* can parse the graph6 format in a human readable form, such as an adjacency matrix. The format does not represent the actual embedding of the graph, therefore it only encodes the abstract structure of the graphs.

3. *Sparse*6: This format is similar to graph6, and was also invented by McKay. It is optimized for sparse graphs (e.g. cubic graphs with 20 or more vertices [19]).

The program plantri can be included in sage. The above output formats are implemented in sage, such that the generated graphs of plantri can be directly parsed in sage internal graph objects. This in particular is very useful, since the automated conjecture-making tool, which we are going to use, can be installed in sage. Therefore, we do not need to worry about compatibility.

Example 4.24. Given a graph $G = (V, E)$ with $n = 5$ vertices. The graph embedding is visualized in figure 4.18.

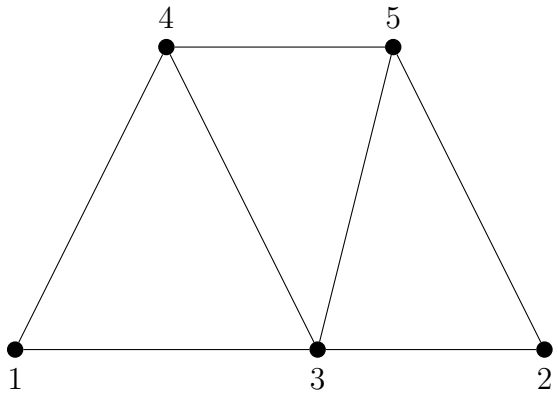


Figure 4.18: Planar graph with $n = 5$ vertices for demonstrating planar codes.

The resulting planar code is then given by

$$\underbrace{5}_{\text{n}}\ \underbrace{340}_{1}\ \underbrace{350}_{2}\ \underbrace{14520}_{3}\ \underbrace{1530}_{4}\ \underbrace{2340}_{5} \tag{91}$$

The code starts with the number of vertices of the graph, which is $n = 5$. Then, the code can be splitted into 5 sections, each section represents one vertex, sorted in ascending order. For example: vertex 1 has neighbors $3, 4$ (clockwise order). Each section ends with a 0.

4.2.3 Usage of plantri in sage

In the following we are going to setup the sage environment and install plantri on a linux machine (more specifically Ubuntu 18.04. LTS[20]). Then, we are going to show how to use plantri in sage and do a comparison of the execution times.
We had troubles installing additional sage packages in sage using the sage installation, which is officially distributed via the "apt" package management of ubuntu. Therefore, we recommend a manual installation. First, we download the sage pre-built binaries from the sage website[21]. Second, we extract the downloaded file with the command, shown in listing 4.

```
1 tar jxf sage-8.9-Ubuntu_18.04-x86_64.tar.bz2
```

Listing 4: Extracting the sage pre-built binaries.

After navigating to the extracted directory, starting sage is rather simple (it might be necessary to grant executable permissions using chmod).

```
1 ./sage
```

Listing 5: Starting the sage environment.

We install the additional plantri package for sage by using the following command.

```
1 ./sage -i plantri
```

Listing 6: Installing plantri for our sage environment.

Furthermore, we can use sage in an interactive browser environment using a *Jupyter* notebook. We start the notebook using the following command.

[20]The procedure is similar on all unix based machines.
[21]https://www.sagemath.org/download.html (visited on 12/29/2019).

```
1 notebook()
```

Listing 7: Starting a jupyter notebook via sage.

Using the above command, a fully functional jupyter notebook, with sage and plantri installed, is started. We can use it by opening a browser and accessing the localhost (http://localhost:8080).
To give an example on how to use plantri with sage and on how they interact with each other, we generate a simple 3-connected quadrangulation with 8 vertices (as we have seen, there is exactly one up to isomorphism) and check if the generated graph is isomorphic to the cube [79].

```
1 gen = graphs.quadrangulations(8, minimum_connectivity=3)
2 g = next(gen)
3 if g.is_isomorphic(graphs.CubeGraph(3)):
4         plot(g)
```

Listing 8: Generating a simple 3-connected quadrangulation with 8 vertices.

In the first line of listing 8, we generate a simple quadrangulation with minimum connectivity of 3 and exactly 8 vertices. The graph generators are structured in the *sage.graphs.graph_generators* module, and are accessed using the *GraphGenerators* class. The used object is called *graphs* [79]. The *next*-function of the generator object returns the first graph of the generation.
We can plot the graph object using the *plot*-function. The output of the above listing is shown in figure 4.19.

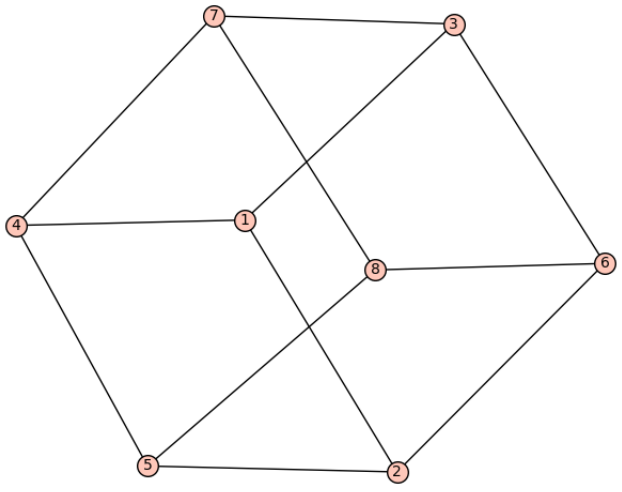


Figure 4.19: Example-output of the plot-function.

Notice, that the plot is going to be different each time one evaluates the plot-function. However, we have some options for changing the layout behavior of the plot-function. In listing 9, we change the layout to “planar”, which causes the plot-function to avoid crossing edges. The resulting output is shown in figure 4.20.

```
1 plot(g, layout="planar")
```

Listing 9: Changing the layout of the plot-function to planar.

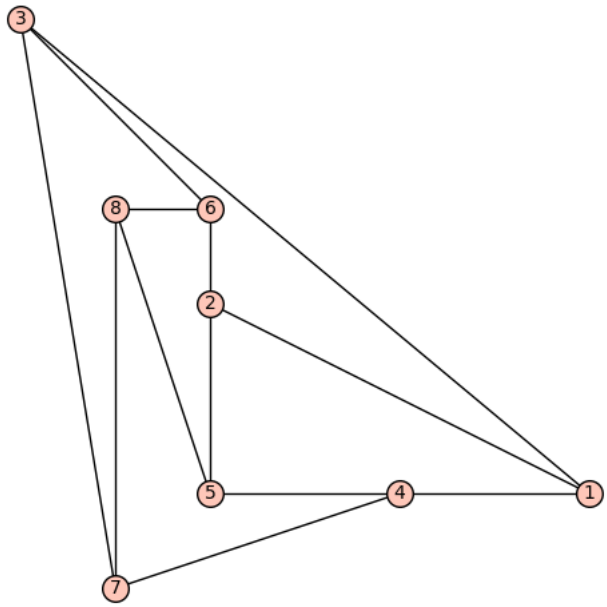


Figure 4.20: Changing the layout of the plot-function to planar.

4.2.4 Runtime performance comparison

In the following, we are going to compare the execution times of the generator function of the integrated version of plantri in sage with the "plain" version of plantri. We are going to generate simple 3-connected quadrangulations, starting with the plain version of plantri. All calculations are performed on a machine with an Intel core i5-4570 with 4-cores at 3.20GHz and 16GB of RAM. We write a small script, in order to call plantri automatically with different parameters.

```
1 #!/bin/bash
2 for i in $(seq 8 1 30)
3 do ./plantri -q -c3 $i > /dev/null; done
```

Listing 10: Bash script for calling plantri with different parameters.

All 3-connected ($-c3$) simple quadrangulations ($-q$), with 8 up to 30 vertices (l. 2), are generated. Since we are only interested in the execution times at the moment, we send the output of plantri to */dev/null*.

Graph count	**Vertex count**	**CPU time (seconds)**
1	8	0.00
1	10	0.00
1	11	0.00
3	12	0.00
3	13	0.00
11	14	0.00
18	15	0.00
58	16	0.00
139	17	0.00
451	18	0.00
1326	19	0.00
4461	20	0.01
14554	21	0.04
49957	22	0.15
171159	23	0.49
598102	24	1.70
2098675	25	5.87
7437910	26	20.67
26490072	27	72.51
94944685	28	255.02
341867921	29	904.37
1236864842	30	3253.92

Table 4.1: Execution times of plantri (1st run).

We perform four runs for each data point. The results of the first run are shown in table 4.1. The other three runs can be found in A.3 in the appendix. The data points of the first run are plotted in figure 4.21.

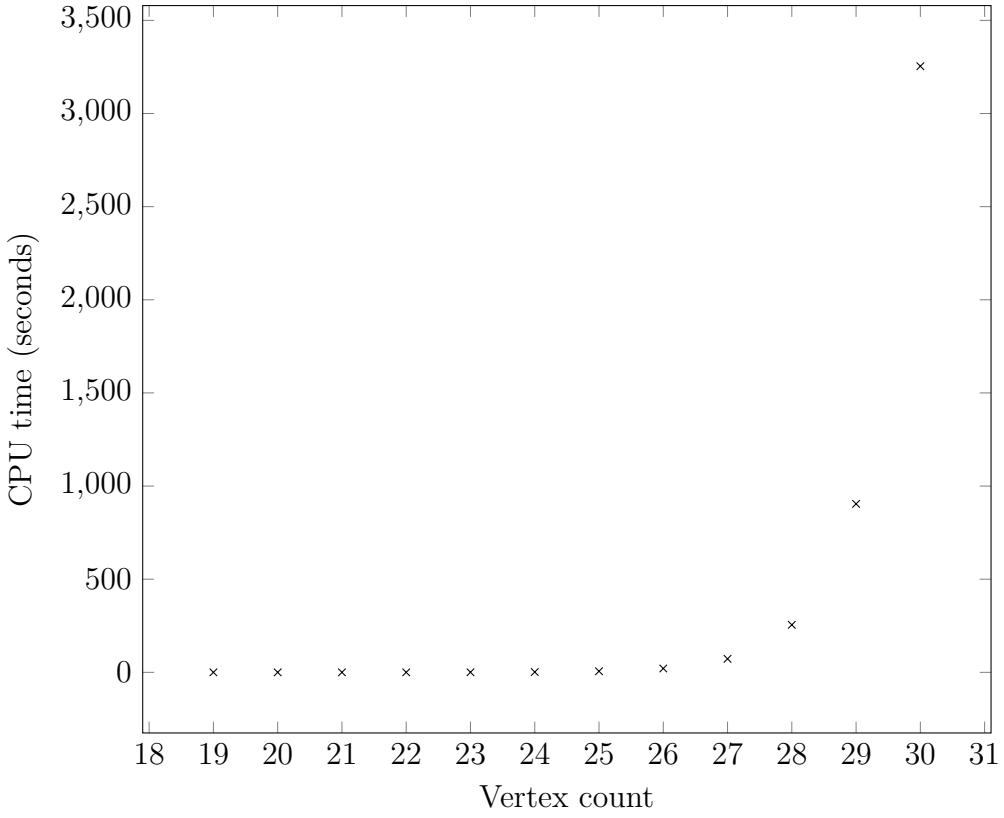


Figure 4.21: Plot: execution times of plantri.

Writing the output of the generator to stdout or to a file takes a lot of time. For that reason, we are going to measure the execution time of the plain generation without writing any output. Since plantri does not provide any flag to disable its output completely, we modify the source code. We open the "plantri.c" source file, and delete the following two lines in the method "got_one" (line 4261).

```
1 if (dswitch) (*write_dual_graph)(outfile,doflip);
2 else         (*write_graph)(outfile,doflip);
```

Listing 11: Disabling the output of plantri.

Furthermore, we modify our script, shown in listing 10, in such a manner that it does not send the output of plantri to */dev/null* (delete "> */dev/null*" in line 4). Next, we need to recompile the plantri.c source file, using the GCC, as shown in listing 2. Finally, we start the script again. The results are shown in table 4.2.

Graph count	Vertex count	CPU time (seconds)
1	8	0.00
1	10	0.00
1	11	0.00
3	12	0.00
3	13	0.00
11	14	0.00
18	15	0.00
58	16	0.00
139	17	0.00
451	18	0.00
1326	19	0.00
4461	20	0.01
14554	21	0.04
49957	22	0.13
171159	23	0.43
598102	24	1.47
2098675	25	5.06
7437910	26	17.70
26490072	27	61.60
94944685	28	216.31
341867921	29	766.02
1236864842	30	2726.30

Table 4.2: Execution times of plantri without generating any output.

It is obvious, that the runtime difference is more significant, if more graphs are generated (compare the runtime with 30 vertices), since more data is forwarded to stdout.

Next, we are going to perform the same test using our sage environment. We perform the generation with the following code.

```
1  import timeit
2  import functools
3  def gen_test(n):
4      gen = graphs.quadrangulations(n, minimum_connectivity=3)
5      for _ in gen: pass
6      pass
7  for n in range(8,31):
8      elapsed_time =
9          timeit.timeit(functools.partial(gen_test, n),number=4)
10     print("vertex count ",n, "time ", elapsed_time)
```

Listing 12: Generating all 3-connected simple quadrangulations and measuring the execution time using sage.

In the following, we are going to explain the code in listing 12, since it contains some specialities. We are using the *timeit* module for measuring the execution time (l.1). It is going to pick the best way of measuring time on your machine and is going to run our code snippet as often as we want using the *number*-parameter (l.9). We run the generation 4 times. The object *graphs* of the *sage.graphs.graph_generators*-module returns a generator for the demanded graph family (in this case for the 3-connected simple quadrangulations with a specific number of vertices n) (l.4). In order to generate all graphs of the demanded class, e.g. all 3-connected simple quadrangulations of order 20, we need to loop through the generator. After each iteration step, a new quadrangulation is generated[22]. Since we are not interested in the output at the moment, we can use the *pass*-statement (l.5), which is a *null* operation [40]. Next, we pass the defined function *gen_test* to the *timeit* function. Since we want to pass the number of vertices to our function, we need to use closures. Therefore, we wrap our function in another function using the function *partial* of the *functools* module (l.9). The behavior of the *timeit* module in our case is discussed in [53]. Finally, we execute our code snippet using *timeit* with 8 up to 30 vertices, and perform a total of four runs. The execution times are shown in table 4.3. The data points of the first run are plotted in figure 4.22.

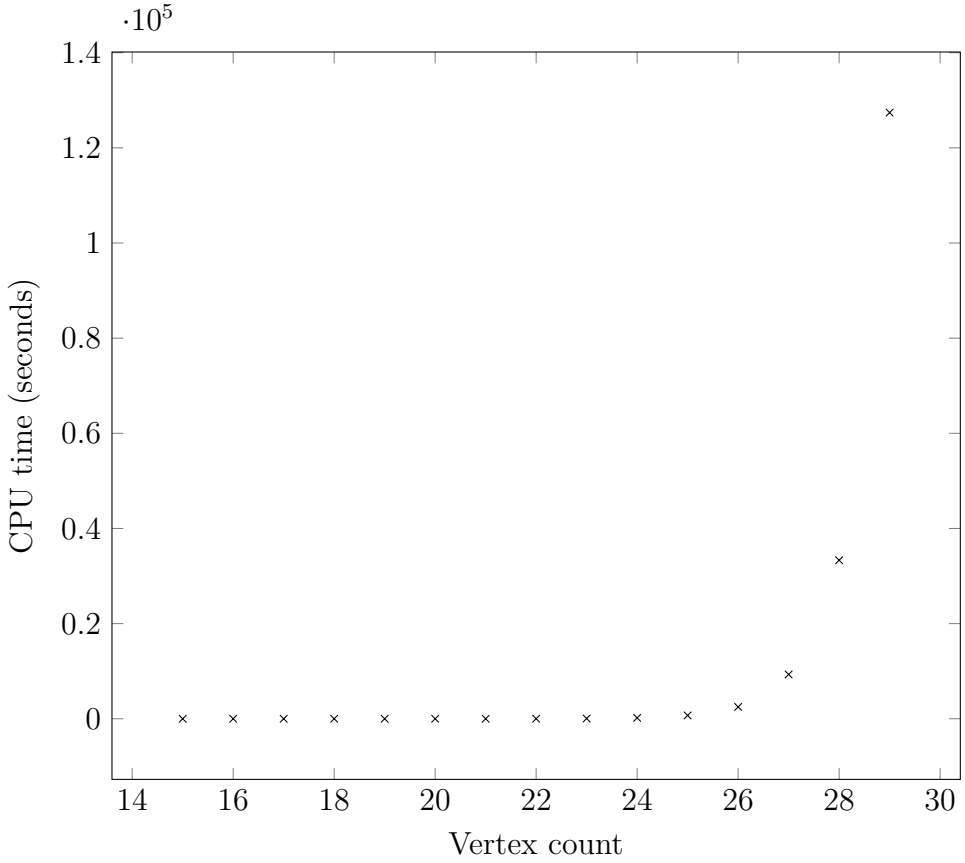


Figure 4.22: Plot: execution times of plantri in sage.

[22]Notice, that parsing the generator into a list by passing it to the *list* constructor, as suggested in [79], results in a system crash for a vertex count of 25 and above due to memory overflow.

Graph count	Vertex count	CPU time (seconds)
1	8	0.01
1	10	0.01
1	11	0.01
3	12	0.01
3	13	0.01
11	14	0.01
18	15	0.01
58	16	0.02
139	17	0.04
451	18	0.11
1326	19	0.33
4461	20	1.15
14554	21	3.94
49957	22	14.13
171159	23	50.79
598102	24	187.32
2098675	25	703.74
7437910	26	2491.38
26490072	27	9315.46
94944685	28	33326.80
341867921	29	127407.80
1236864842	30	–

Table 4.3: Execution times of plantri using sage (1st run).

The other three runs can be found in A.4 in the appendix. The execution times are drastically increased in comparison to the plain version of plantri. There are many possible reasons, e.g. there could be an overhead in the way sage calls external code (in this case plantri) or parsing the planar code into sage graph objects could increase the complexity. Furthermore, we do not have any detailed knowledge of the internal structure of the graph generators in sage or of the generator objects of python in general. A bottleneck could be conceivable. However, we will not conduct a detailed examination at this point.

5 Generation of optimal 1-planar graphs

In the following section, we are going to develop an algorithm for generating optimal 1-planar graphs. The investigations, made in the previous sections, especially in section 3.2 and in section 4.2, build the foundation for the implementation.
We have already discovered that the skeletons of optimal 1-planar graphs are the simple quadrangulations. The optimal 1-planar graphs are obtained by adding the crossing edges in each quadrangular face of their skeletons. In the work of Suzuki, a generation theorem for optimal 1-planar graphs, using nearly the same expansion rules as Brinkmann et al. for generating the simple quadrangulations, is given and proven. In the work of Brandenburg an algorithm for recognizing optimal 1-planar graphs in linear time is presented. He uses a reduction system, containing the P_1-reductions, which he calls Schumacher reduction (SR), and P_3-reductions, which he calls Crossed cube reduction (CR). Using their results and the investigations we have made in the previous sections, we have strong evidence that optimal 1-planar graphs can be efficiently generated using the expansion rules for simple quadrangulations. Therefore, we consider the following four possibilities for the implementation of the algorithm.

1. Extending the source code of *plantri* by implementing a method, which adds the crossing edges in each quadrangular face after an expansion rule is applied.

2. Extending *plantri* by implementing a *filter*-plugin, which allows to make changes to a quadrangulation after it has been generated. One could add the crossing edges to each face after the generation, instead of changing the expansion rules.

3. Using a sage environment with plantri. Generate the simple quadrangulations, parse them directly into sage graph objects, and finally add the crossing edges in each face.

4. Implementing the whole algorithm from the beginning.

In the following, we are going to evaluate the different options for the implementation and discuss the advantages and disadvantages. The first option is most likely the most efficient one from a computational point of view. [P]lantri has been well tested over the last years and has become a de facto standard for planar graph generation. Also, the isomorph rejection, using the program *nauty* by McKay and Piperno, is the best known and most up-to-date. However, plantri is a C program with approximately $20,000$ lines of code in a single C-file with very sparse documentation. Extending the source code directly would require us to understand the code structure first, which would, most likely, be very time consuming. Furthermore, there are no automated tests for plantri (at least no published ones), hence there is no publicly available test coverage. Therefore, changing the code is dangerous from a software engineering point of view and it is likely that any changes of the source code quickly lead to errors.

The second option require us to implement a so called *filter* plugin. [P]lantri provides the possibility to implement extensions, called *plugin*s, which has an advantage over changing the source code directly: we do not change any existing code, we just integrate the new code using the plugin system. The plantri user manual gives a detailed description on the different types of plugins and on how to implement them. Also, they provide a few examples for each plugin type. There are two options for solving our problem, using the plugin system of plantri. The first one is to implement a so called *filter**-plugin, which allows to make changes to the generated graph after each expansion. However, we can not add the crossing edges to the internal data structure, since it would break the next expansion[23]. For solving this problem, we could keep a copy of the graph in our own data objects and make changes to them. However, there are certain disadvantages using this approach. First, this would not be very memory efficient. Second, updating the copied version of the graph is error prone.

Another plugin related option would be to implement a so called *filer*-plugin. A filter, not to be mistaken with *filter**, allows us to make changes to the finished, generated graph. For example, we could add the crossing edges in each face. Nevertheless, fundamental knowledge of the internal code structure of plantri must be known for this as well.

The third option require us to set up a sage environment, install plantri in it and use the sage graph objects to add the crossings to the generated graphs. This solution has several advantages and disadvantages. First of all and as we have seen in the previous section, the implementation would not be very efficient from a computational point of view. Furthermore, for this solution it is most likely that the most memory is consumed. Nevertheless, we are able to use the internal graph library of sage, which provides a lot of useful methods for manipulating graphs. Additionally, one main requirement for our generator is, that the output can easily be processed to an automated graph conjecture-making tool. We are going to use the tool *conjecturing*, which we are going to present in a later chapter. Conjecturing can also be easily integrated in a sage environment. Therefore, using sage for the graph generation as well has obvious advantages.

The last option would be an implementation from scratch, meaning we implement the expansion rules given by Suzuki. As we are going to see in the next section, the expansion system by Suzuki does not really provide any new results. He uses exactly the same expansion and reduction rules as given by Brinkmann et al. and extends them by adding and/or removing crossing edges. From a computational standpoint, this procedure would result in nearly the same complexity, compared to generating the simple quadrangulations and adding the crossing edges afterwards. Furthermore, the output of the generator should be compatible with the program nauty for graph isomorph rejection, because implementing

[23]This would happen, because we would change the cyclic order of the edges around the vertices, or simply because the resulting graph would leave its graph family (which are the simple quadrangulations in this case).

the canonical construction path method, as invented by McKay and Piperno, would most likely consume more time than implementing the graph generation itself.
Furthermore, nauty and plantri are around for several years, and therefore they are well tested and known to be efficient. Therefore, our algorithm is going to be based on plantri, which we are going to use to generate the skeletons of the optimal 1-planar graphs, which are the simple quadrangulations. Additionally, we are going to set up plantri in a sage environment, since the output can easily be processed to the automated conjecture-making tool.

However, in the next section we are first going to present the expansion rules given by Suzuki in order to show the similarity between the expansion system given by Brinkmann et al. Then, we are going to present our algorithm for generating optimal 1-planar graphs, and finally implement it using a sage environment.
Last but not least, we are going to test our implementation, give an example of the output and finally execute a simple performance test.

5.1 Expansion rules

We have shown in the last sections that the skeletons of optimal 1-planar graphs are the simple quadrangulations. Suzuki defines two expansions, and their inverses, called reductions. Let G be an optimal 1-planar graph and let f be a face of the skeleton, which is bounded by a 4-cycle $v_1v_2v_3v_4$.

Definition 5.1. [97, p.7](Q_f*-contraction,* Q_v*-splitting*) A Q_f*-contraction* at $\{v_1, v_3\}$ is defined as follows:

(i) Delete the pair of crossing edges $\{(v_1, v_3), (v_2, v_4)\}$.

(ii) Identify v_1 and v_3 and replace the two pairs of multiple edges $\{(v_1, v_2), (v_3, v_2)\}$ and $\{(v_1, v_4), (v_3, v_4)\}$ with two single edges respectively.

The inverse operation of the Q_f-contraction is said to be the Q_v*-splitting.* If the operations break the simpleness of the graph, then we do not apply them.

Example 5.2. Given an optimal 1-planar graph $G = (V, E)$. Let f be a face of the sekeleton of G, which is bounded by $C_4 = v_1v_2v_3v_4$. The excerpt of the graph, containing f, is shown in figure 5.1. In order to perform a Q_f-contraction at $\{v_1, v_3\}$, we first remove the two crossing edges $\{(v_1, v_3), (v_2, v_4)\}$. The resulting graph is shown in figure 5.2. Now, performing step (ii) of the Q_f-contraction at $\{v_1, v_3\}$ is excactly the same as a P_1-contraction, which we have defined in section 4.2. We replace the vertices v_1, v_3 with a single vertex v_x, and consequently the edges $(v_1, v_4), (v_3, v_4)$ and $(v_1, v_2), (v_3, v_4)$ with the single edges (v_x, v_4) and (v_x, v_2). The resulting graph is shown in figure 5.3.

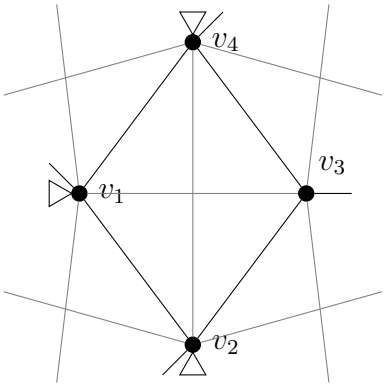


Figure 5.1: Excerpt of graph G for presenting a Q_f-contraction.

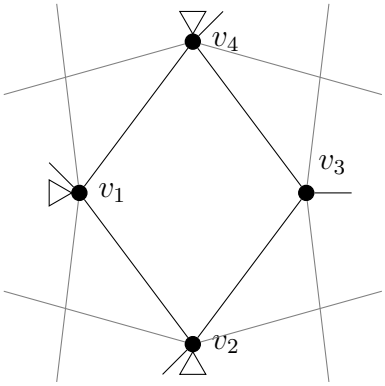


Figure 5.2: G after step (i) of a Q_f-contraction.

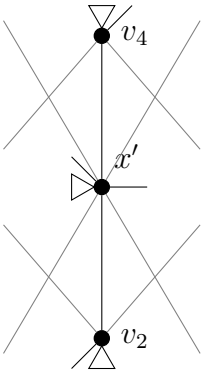


Figure 5.3: G after step (ii) of a Q_f-contraction.

Definition 5.3. [97, p.7](Q_4-*addition,* Q_4-*removal*) A Q_4-*addition* is an expansion which is performed as follows:

(i) Delete a pair of crossing edges $\{(v_1, v_3), (v_2, v_4)\}$.

(ii) Add a 4-cycle $u_1u_2u_3u_4$ inside the interior of f and join v_i and u_i for $i = 1, 2, 3, 4$.

(iii) Add a pair of crossing edges to each face bounded by a 4-cycle consisting of non-crossing edges.

The inverse operation (reduction) of a Q_4-addition is called the Q_4-*removal*. If the operations break the simpleness of the graph, then we do not apply them.

Example 5.4. Given an optimal 1-planar graph $G = (V, E)$. Let f be a face of the skeleton of G, which is bounded by $C_4 = v_1v_2v_3v_4$. The embedding of G is shown in figure 5.4. Step (i) of a Q_4-addition removes the crossing edges $(v_1, v_3), (v_2, v_4)$.

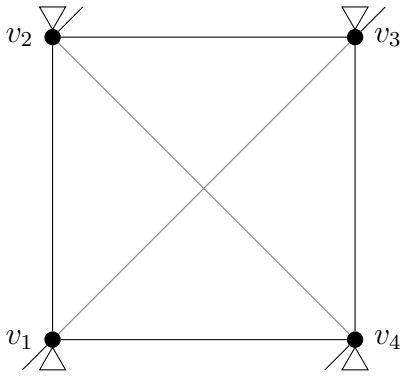


Figure 5.4: Initial state of G.

In step (ii), a 4-cycle $C_4 = u_1u_2u_3u_4$ is added in the interior of the graph, which has resulted after step (i). First, we add the edges $(v_1, u_1), (v_2, u_2), (v_3, u_3), (v_4, u_4)$, and second, we add two crossing edges in each of the resulting faces (step (iii)). The final graph is shown in figure 5.5. Notice, that step (ii) is exactly the same as a P_3-expansion.

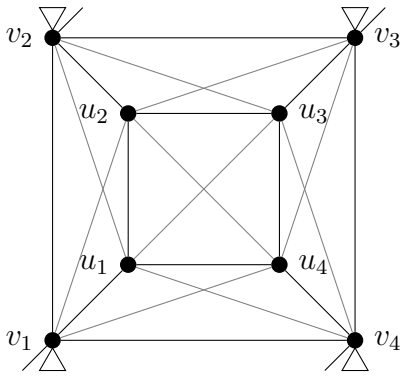


Figure 5.5: G after step (ii) and (iii) of a Q_4-addition.

The expansion rules, which we have introduced in definition 5.1 and 5.3 as given by Suzuki are very similar to the ones given in section 4.2. In fact, both expansions remove the crossing edges, before the actual expansion is carried out. In a final step, the crossing edges are added in each face. There is no real difference, compared to the generation of a simple quadrangulation and adding the crossing edges in each face afterwards. It seems even more complex to do it the other way around. Additionally, in the work of Suzuki, a generation theorem, using the above expansions, is presented, which he proves by using the results of the generation theorem given by Brinkmann et al.
In conclusion, it can be stated that the generation of the optimal 1-planar graphs can be implemented with the expansion rules for simple quadrangulations by inserting two crossing diagonal edges into the interior of each face.

5.2 Implementation of the generation algorithm

In the following, we are going to implement the generation of optimal 1-planar graphs, using the program *plantri* for the generation of the planar skeletons in the sage environment, which we have set up in an earlier section.
We are going to generate the simple quadrangulations, and then add the crossing diagonal edges in each face. Then, we are going to test our implementation, and verify its correctness by using certain properties of optimal 1-planar graphs.
In listing 13 the main method for the generation is shown. It takes the number of vertices n as an input, and returns a collection of all optimal 1-planar graphs of order n. The size of the returned collection coincides with the number of all possible simple quadrangulations of order n.

```
1 def generateOptimal1PlanarGraphs(n):
2     skeletons = generateSkeletons(n)
3     optimal1PlanarGraphs = []
4     for skeleton in skeletons:
5         o1p = addCrossingEdgesToFaces(skeleton)
6         assertGraphIsOptimal1Planar(o1p, n)
7         optimal1PlanarGraphs.append(o1p)
8     return optimal1PlanarGraphs
```

Listing 13: Generation method for optimal 1-planar graphs.

First, the planar skeletons, which are the simple quadrangulations, are generated, using the *generateSkeletons*-method (l. 2). The method returns a collection, containing the planar skeletons. We assign that collection to the variable *skeletons*.

Then, an empty collection is initialized (l. 3), which is going to hold the generated optimal 1-planar graphs. Next, we iterate over the skeleton collection, and for each skeleton the method *addCrossingEdgesToFaces* is called (l. 5). The method takes a graph as an input, which is, in this case, a simple quadrangulation Q, and adds the crossing diagonals in each of the faces of Q. Notice, that the method *addCrossingEdgesToFaces* does not check if the provided input graph is a valid quadrangulation, hence the result is not necessarily a valid optimal 1-planar graph. However, since we are using plantri for the generation of the skeletons, we assume that the results of the used methods are correct.
Therefore, the method returns an optimal 1-planar graph, which is assigned to the variable *o1p* (l. 5). The method *assertGraphIsOptimal1Planar* validates if the graph is optimal 1-planar by using the upper bound for the number of edges. Notice, that this check could be considered as weak, since fulfilling the upper bound does not necessarily mean that a given graph is optimal 1-planar. However, since we are sure that the planar skeletons are correct, and we have proven that adding the crossing diagonals in each of the faces leads to optimal 1-planarity, the test for the upper bound rather prevents profound implementation errors.

In the following, we are going to present the methods used in listing 13. The method *generateSkeletons* takes the desired graph order n as an input and returns a generator for simple 3-connected quadrangulations of order n. The method uses the integrated version of plantri, as we have shown in section 4.2.3.

```
1 def generateSkeletons(n):
2     gen = graphs.quadrangulations(n, minimum_connectivity=3)
3     return gen
```

Listing 14: Generating the planar skeletons.

The method *addCrossingEdgesToFaces* is shown in listing 15. It takes a graph as an input and adds the crossing diagonal edges in each of its faces. To do so, the method iterates over the faces of the input graph q (l. 2). Sage provides a convenient method for getting the faces of a given graph, which is simply called *faces*. Four variables are initialized, namely *temp*, v_1, v_2 and *count*. Next, we iterate over each edge of each face (l. 7). In the first iteration step, the first edge of the first iteration step is stored in the variable *temp*. v_1 and v_2 are its ends. After the assignment, the iteration over the edge set of a face is proceeded (l. 12). We assign the ends of the next edge of the iteration to the variables v_x and v_y (ll. 13-14). Next, we check if v_x or v_y is the opposite vertex of v_1 or v_2. If the result is positive, we insert an edge between them and the *count* integer variable is incremented. If the variable *count* is equal to 2, we stop the iteration over the edge set of a face (ll. 18-19), since we need exactly two diagonals in each quadrangular face.

```
1  def addCrossingEdgesToFaces(q):
2      for f in q.faces():
3          temp = None
4          v_1 = None
5          v_2 = None
6          count = 0
7          for e in f:
8              if temp == None:
9                  temp = e
10                 v_1 = e[0]
11                 v_2 = e[1]
12                 continue
13             v_x = e[0]
14             v_y = e[1]
15
16             if addEdge(v_1,v_2,v_x,v_y,q):
17                 count+=1
18             if count == 2:
19                 break
20     return q
```

Listing 15: Adding the crossing diagonal edges.

```
1  def addEdge(v_1,v_2,v_x,v_y,q):
2      if v_1 != v_x and not q.has_edge(v_1,v_x):
3          q.add_edge(v_1,v_x)
4          return True
5      elif v_2 != v_x and not q.has_edge(v_2,v_x):
6          q.add_edge(v_2,v_x)
7          return True
8      elif v_1 != v_y and not q.has_edge(v_1,v_y):
9          q.add_edge(v_1,v_y)
10         return True
11     elif v_2 != v_y and not q.has_edge(v_2,v_y):
12         q.add_edge(v_2,v_y)
13         return True
14     return False
```

Listing 16: The *addEdge*-method.

The *addEdge*-method is shown in listing 16. It takes four vertices and the corresponding graph object as an input[24]. The method checks adjacency for each combination of v_1, v_2, v_x and v_y. If two vertices vertices are not adjacent, an edge between them is added and the method returns *true*.

```
1 def assertGraphIsOptimal1Planar(g):
2     numEdges = g.size()
3     n = g.order()
4     bound = 4*n-8
5     if numEdges != bound:
6         raise Exception("Graph is not optimal 1-planar")
```

Listing 17: Assertion for the upper bound for testing optimal 1-planarity.

In listing 17 the *assertGraphIsOptimal1Planar* method is shown. It takes a graph g as an input checks whether g has exactly $4n - 8$ edges, whereby n is the order of g (l.3). The *size*-method of the graph object returns the number of edges (l.2). If the bound is not satisfied, an exception is thrown (l.5).

In the following, we are going to give two examples of the presented implementation by generating an optimal 1-planar graph with 8 and with 14 vertices.

Example 5.5. We print the edge set of the generated skeleton with $n = 8$ vertices by adding the following statement after line 4 of listing 13.

```
1 print(skeleton.edges())
```

Listing 18: Print-statement for the edge set of the generated skeleton.

Setting $n = 8$ results in the following output of the above statement:

```
1 (1, 2, None), (1, 3, None), (1, 4, None), (2, 5, None),
2 (2, 6, None),(3, 6, None), (3, 7, None), (4, 5, None),
3 (4, 7, None), (5, 8, None),(6, 8, None), (7, 8, None)
```

The vertices are labeled with integer numbers. Following the generally known definitions, the edges are given in round brackets, e.g. $(1, 2)$ means that there is an edge between vertex 1 and 2. The third parameter represents an edge label and is optional. Since we have not set any edge labels, the third parameter is *None*. We label the edges in ascending order, according to the above output, with e_i, whereby $i = 1, \cdots, 12$.

[24]Notice, that the method does not check if the provided vertices are assigned to the provided graph. Hence, the user of the method must ensure this.

The rotation system $\Pi = \{\pi_1, \cdots, \pi_n\}$, whereby π_k is the rotation of the edges around vertex k in clockwise order with $k = 1, \ldots, n$, is given by

$$\pi_1 = \begin{pmatrix} e_1 & e_2 & e_3 \\ e_2 & e_3 & e_1 \end{pmatrix} \qquad \pi_2 = \begin{pmatrix} e_1 & e_4 & e_5 \\ e_4 & e_5 & e_1 \end{pmatrix} \qquad \pi_3 = \begin{pmatrix} e_2 & e_6 & e_7 \\ e_6 & e_7 & e_2 \end{pmatrix} \tag{92}$$

$$\pi_4 = \begin{pmatrix} e_3 & e_8 & e_9 \\ e_9 & e_3 & e_8 \end{pmatrix} \qquad \pi_5 = \begin{pmatrix} e_4 & e_8 & e_{10} \\ e_8 & e_{10} & e_4 \end{pmatrix} \qquad \pi_6 = \begin{pmatrix} e_5 & e_6 & e_{11} \\ e_{11} & e_5 & e_6 \end{pmatrix} \tag{93}$$

$$\pi_7 = \begin{pmatrix} e_7 & e_9 & e_{12} \\ e_{12} & e_7 & e_9 \end{pmatrix} \qquad \pi_8 = \begin{pmatrix} e_{10} & e_{11} & e_{12} \\ e_{12} & e_{10} & e_{11} \end{pmatrix} \tag{94}$$

The embedding is shown in figure 5.6.

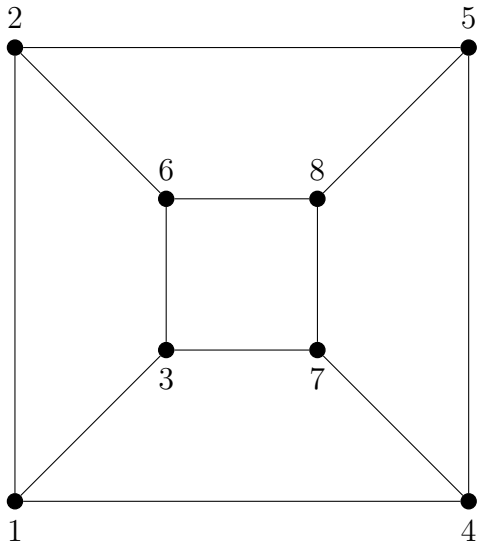


Figure 5.6: Embedding of the planar skeleton with $n = 8$ vertices.

Next, we print the edges of the optimal 1-planar graph by calling the *generateOptimal1PlanarGraphs* method and printing the edges of the resulting graph. The statements are shown in listing 19.

```
1 o1ps = generateOptimal1PlanarGraphs(8)
2 if len(o1ps) == 1:
3     o1p = o1ps[0]
4     print(o1p.edges())
```

Listing 19: Generating an optimal 1-planar graph with 8 vertices and printing its edge set.

The following output is produced, whereby the crossing edges are highlighted in bold.

```
1 (1, 2, None), (1, 3, None), (1, 4, None), (1, 5, None),
2 (1, 6, None), (1, 7, None), (2, 3, None), (2, 4, None),
3 (2, 5, None), (2, 6, None), (2, 8, None), (3, 4, None),
4 (3, 6, None), (3, 7, None), (3, 8, None), (4, 5, None),
5 (4, 7, None), (4, 8, None), (5, 6, None), (5, 7, None),
6 (5, 8, None), (6, 7, None), (6, 8, None), (7, 8, None)
```

Notice that there are excactly $2(n-2) = 12$ crossing edges (compare with the results of section 3.2). The embedding of the resulting optimal 1-planar graph is shown in figure 5.7. The crossing edges are highlighted in grey color.

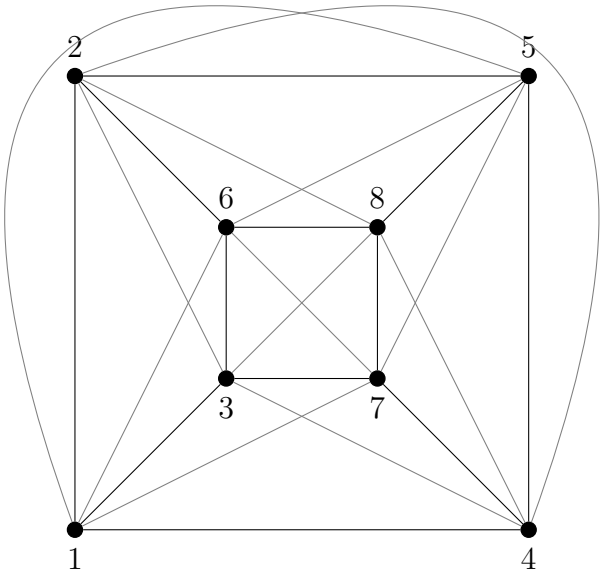


Figure 5.7: Resulting optimal 1-planar graph.

Example 5.6. In the following we are going to generate an optimal 1-planar graph with $n = 14$ vertices. There are exactly 11 optimal 1-planar graphs of order 14. We pick one randomly. The skeleton of the selected graph is shown in figure 5.8.
The edge set of the resulting optimal 1-planar graph is given by the following listing, whereby the crossing edges are highlighted in bold.

```
1  (1, 2, None), (1, 3, None), (1, 4, None), (1, 5, None),
2  (1, 6, None), (1, 8, None), (2, 3, None), (2, 4, None),
3  (2, 5, None), (2, 6, None), (2, 10, None), (3, 4, None),
4  (3, 6, None), (3, 7, None), (3, 8, None), (3, 11, None),
5  (3, 12, None), (4, 5, None), (4, 8, None), (4, 9, None),
6  (5, 6, None), (5, 8, None), (5, 9, None), (5, 10, None),
7  (5, 13, None), (6, 7, None), (6, 10, None), (6, 11, None),
8  (6, 14, None), (7, 8, None), (7, 11, None), (7, 12, None),
9  (7, 14, None), (8, 9, None), (8, 12, None), (8, 13, None),
10 (9, 10, None), (9, 12, None), (9, 13, None), (10, 11, None),
11 (10, 12, None), (10, 13, None), (10, 14, None), (11, 12, None),
12 (11, 14, None), (12, 13, None), (12, 14, None), (13, 14, None)
```

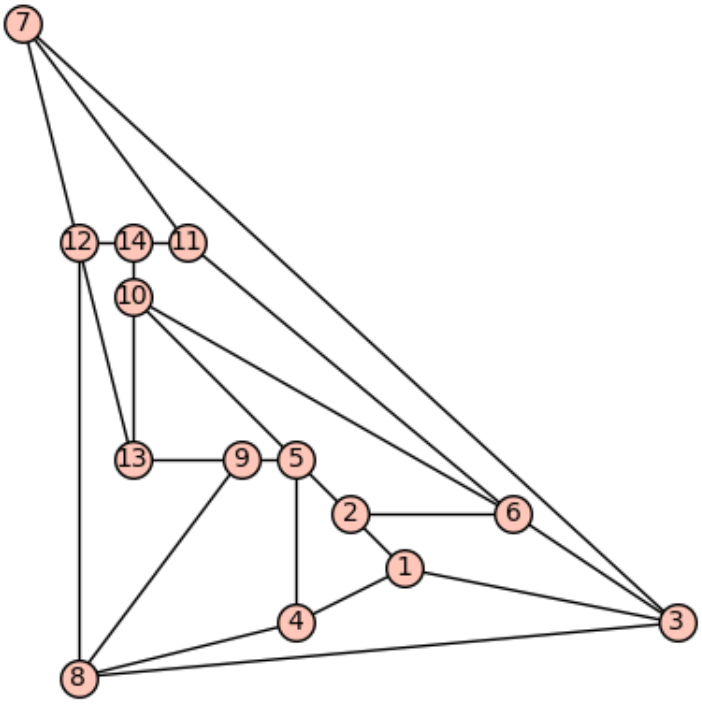


Figure 5.8: Skeleton of randomly picked optimal 1-planar graph with $n = 14$ vertices.

5.3 Runtime performance test

In the following we are going to perform a runtime performance test for our implementation using the methods of the *timeit* module, which we have introduced in section 4.2.4. We are going to generate all optimal 1-planar graphs of order $n \in \{8, \cdots, 25\}$ and perform 10 iterations for each n. All calculations are performed on a machine with an Intel core i5-4570 with 4-cores at 3.20GHz and 16GB of RAM.

The time measurements of the first two runs are shown in table 5.1.

Vertex count	CPU time (s)
8	0.01
10	0.01
11	0.01
12	0.01
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.23
19	0.75
20	2.53
21	8.33
22	29.46
23	107.03
24	396.76
25	1437.58

(a) 1st run.

Vertex count	CPU time (s)
8	0.02
10	0.01
11	0.01
12	0.01
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.23
19	0.69
20	2.43
21	8.16
22	29.66
23	106.76
24	392.74
25	1436.50

(b) 2nd run.

Table 5.1: Execution times of the generation of optimal 1-planar graphs (1st, 2nd run).

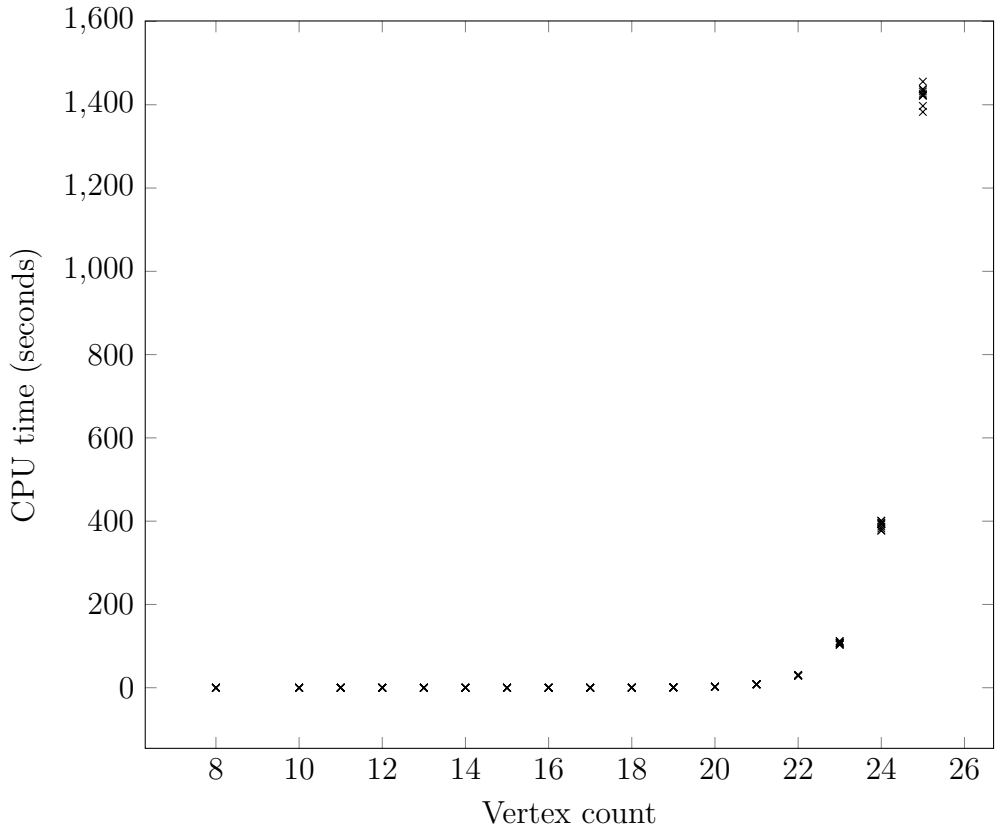


Figure 5.9: Plot: execution times of the implemented generation algorithm for optimal 1-planar graphs.

The other eight iterations can be found in A.5 in the appendix. All results are shown in figure 5.9. We perform a regression analysis. First, we do a curve fitting using sage as shown here [27]. Imagine, that the variable *data* contains all the data points of the 10 iterations of our runtime performance execution. The code is shown in listing 20.

```
1  # imagine 'data' contains all the data points of our
2  performance analysis
3  var('a,b,c,x')
4  model(x) = a^(x-b)+c*x
5  fitted_function = find_fit(data,model)
6  print(fitted_function)
7
8  f(x) = model(a=fitted_function[0].rhs(),
9  b=fitted_function[1].rhs(),c=fitted_function[2].rhs())
10 print(model)
11 print(f(x))
12 # create an empty plot object
13 a = plot([])
14 # add a plot of the model, with respect to x from 0 to 25
15 a += plot(f(x),x,[8.0,25],axes=False)
16
17 # add a plot of the data, in red
18 a += list_plot(data,color='red',axes_labels=['Vertex count',
19 'CPU time (seconds)'],frame=True, fontsize=8)
20 show(a)
```

Listing 20: Fitting the runtime performance data to a function.

First, we create three symbolic variables, named a, b, c and x, using the *var* function (l.3). Next, we create a symbolic expression, called *model*, which is the function to which we want to fit our data (l.4). We choose the following function:

$$f : x \mapsto a^{x-b} + c \cdot x, \quad a, b, c \in \mathbb{R} \tag{95}$$

There is no theoretical reason for the above function, at least nothing that we would have examined, other then the author's intuition. The goal of our analysis is to identify the "best" coefficients a, b and c such that f fits the data well. For that purpose, we call the method *find_fit*, which searches for the best fitting parameters (l.5).

We store its result in the variable *fitted_function*. Finally, we print the calculated coefficients and plot the data points as well as the resulting function (ll.8-20). The plot is shown in figure 5.10, whereby the data points are colored red, and the fitted curve is colored blue.

```
[a == 3.6472775369288275, b == 19.38876211469894, c == 0.005327336742302881]
x |--> c*x + a^(-b + x)
3.6472775369288275^(x - 19.38876211469894) + 0.005327336742302881*x
```

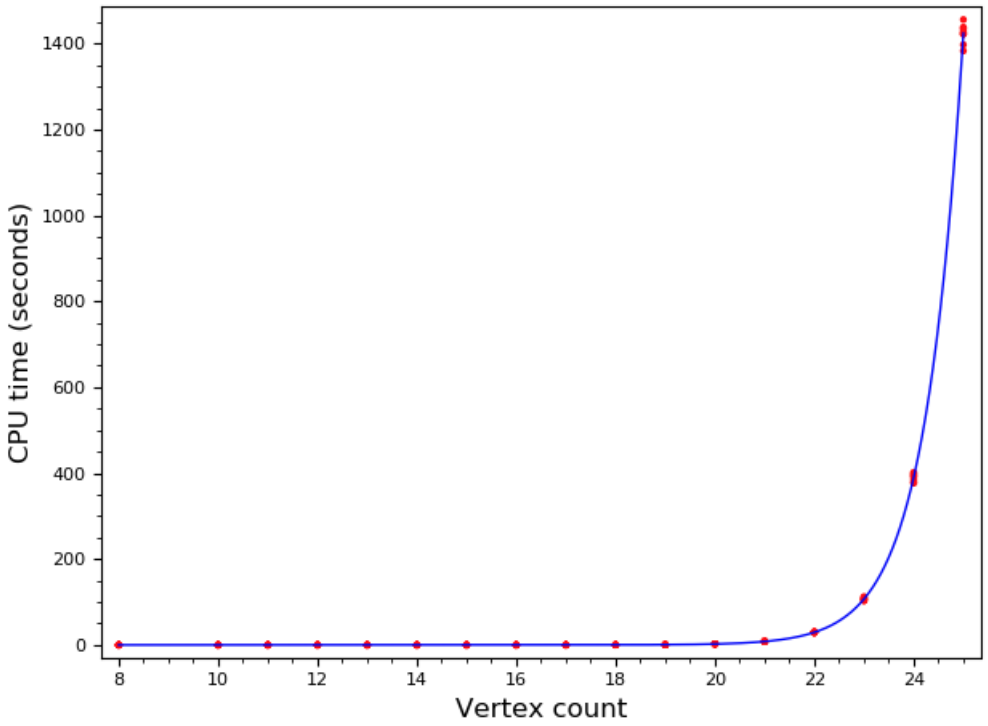


Figure 5.10: Plot: execution times of the generation algorithm for optimal 1-planar graphs and its function model.

In order to analyze the goodness of the fitted function, we calculate the Root Mean Squared Error (RMSE), for which we need to calculate the Sum of Squares Due to Error (SSE) and the Mean Squared Error (MSE). Suppose we have n data points (x_i, y_i). Then, the SSE is given by (cf. [73])

$$SSE = \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{96}$$

The MSE is given by

$$MSE = \frac{SSE}{n} \tag{97}$$

Then, the RMSE is given by

$$RMSE = \sqrt{MSE} \tag{98}$$

We calculate the RMSE using *numpy*, which is a well-known library for python (cf. [72]). The calculation is shown in listing 21. We define a method, called *rmse*, which takes two arrays as an input (l.2) and calculates the RMSE as defined by equation 98. Next, we initialize two lists, named y and *predictions*. Then, we iterate over all data points and append the actual time measurements to the list y and the predictions, made by our model f, to the list *predictions* (ll.6-9). Finally, we calculate the RMSE using the *rmse* function and print its result.

```
1  import numpy as np
2  def rmse(y, predictions):
3      return np.sqrt(((y - predictions) ** 2).mean())
4  y = []
5  predictions = []
6  for dataPoint in data:
7      y.append(dataPoint[1])
8      x = dataPoint[0]
9      predictions.append(f(x))
10 print(rmse(np.asarray(y),np.asarray(predictions)))
```

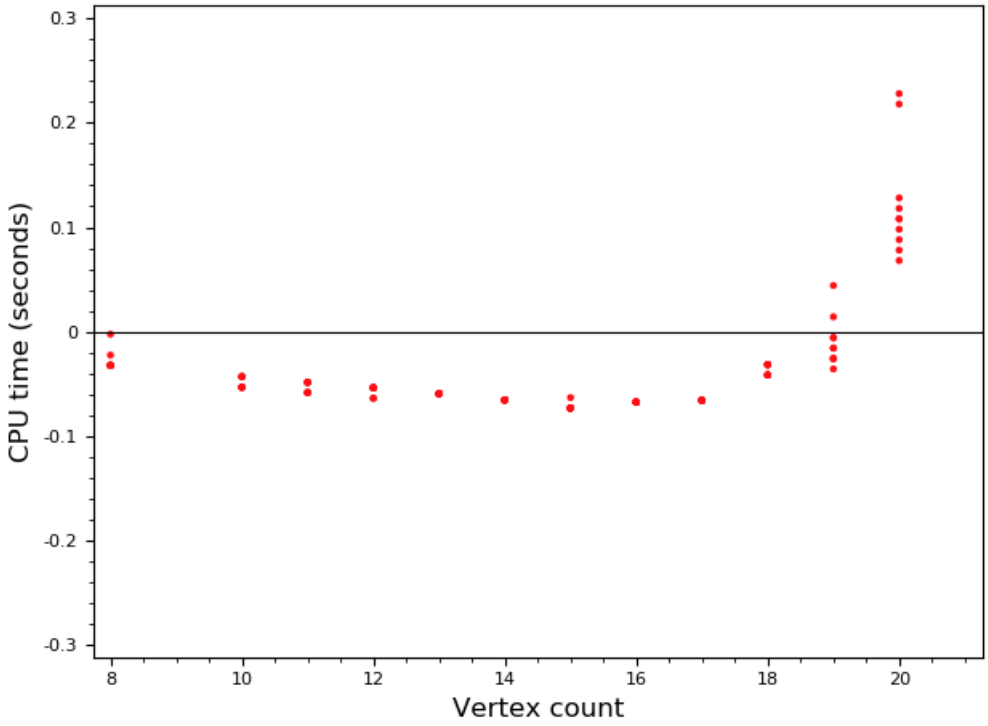
Listing 21: Calculating the RMSE for our model.

The RMSE measures the goodness of our predictive model in comparison to the actual data. In relation to the range of the given data points, a small RMSE indicates a good predictive model. Our data ranges from 0.00 to 1455.69. The RMSE of our model is given by $RMSE = 5.22$, hence the values, predicted by our model, are very close to the actual measurements.

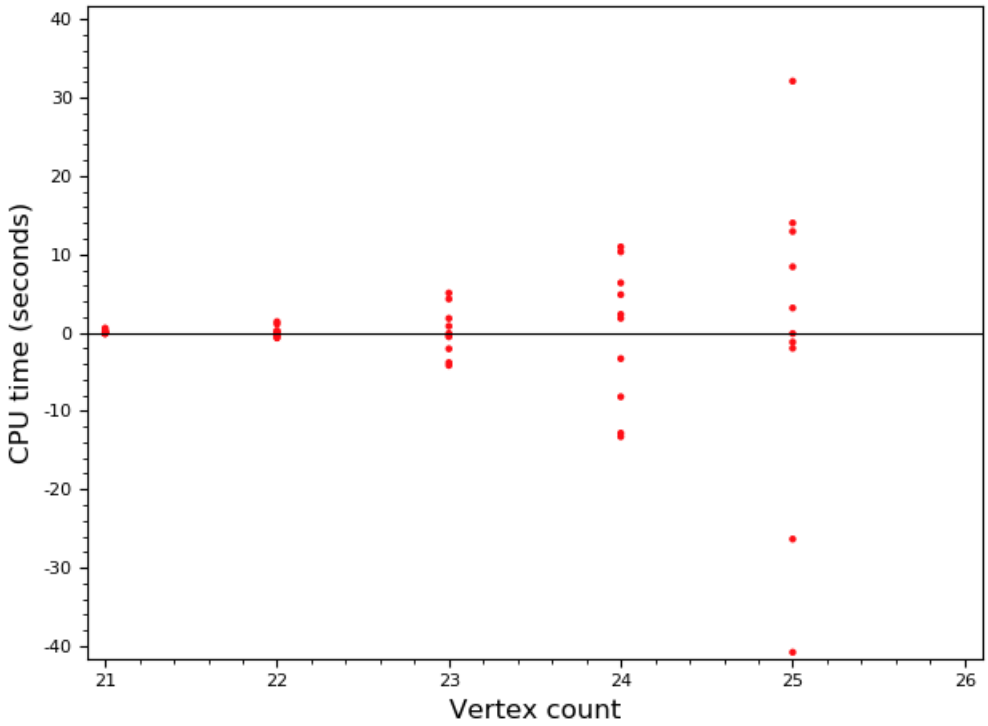
Next, we calculate the residuals and plot them. The used code is shown in listing 22. Since we want to analyze the residuals graphically and the range of the y values is very high, we divide the data set in two quantities, whereby one set contains all data points with x ranging from 8 to 20 and one set contains all data points with x ranging from 21 and above. To do so, we initialize two empty lists (ll. 1). Then, we iterate over all data points and calculate the residuals (ll.2-5). Next, we append all residuals greater than 20 to the first list, and all other residuals to the second list (ll.6-9). Finally, we plot the results (ll.10-15). The plots are shown in figure 5.11.

```
residuals_1,residuals_2 = [],[]
for dataPoint in data:
    x_i = dataPoint[0]
    y_i = dataPoint[1]
    residual = y_i - f(x_i)
    if x_i > 20:
        residuals_1.append((x_i,residual))
    else:
        residuals_2.append((x_i,residual))
residualPlot_1 = plot([])
residualPlot_1 += list_plot(residuals_1,color='red',
axes_labels=['Vertex count', 'CPU time (seconds)'],
frame=True, fontsize=8)
show(residualPlot_1)
residualPlot_2 = plot([])
residualPlot_2 += list_plot(residuals_2,color='red',
axes_labels=['Vertex count', 'CPU time (seconds)'],
frame=True, fontsize=8)
show(residualPlot_2)
```

Listing 22: Calculating the residuals.



(a) Residuals with $7 < x < 21$.



(b) Residuals with $x > 20$.

Figure 5.11: Plot of the residuals.

The residuals are small in terms of the value range, which indicates a good model as well. Now, we can use our model to predict the runtime of our algorithm for larger vertices. For generating all optimal 1-planar graphs with 30 vertices, our model predicts a runtime of approximately 918724 seconds, which are 15312 minutes, 255 hours or 10.6 days. Comparing this result with the runtime of the plain version of plantri for generating the skeletons of the optimal 1-planar graphs, which was 2726 seconds, which are 45.4 minutes and keeping in mind, that we have "just" added the crossing edges in each of the faces, the resulting efficiency is not very promising.
It would be interesting to implement the algorithm directly in plantri, if one is interested in an efficient generation. However, we are more interested in generating a relatively small subset of the optimal 1-planar graphs and forward it to the conjecture-making program, hence runtime performance is not the number one priority.

6 Automated graph conjecture-making

As we have already noted in the introduction of this thesis, a conjecture is a formal mathematical statement, which is initially open to formal proofs. Without keeping automated conjecture-making in mind, we might ask ourselves: what constitutes a good conjecture? Bondy tries to answer this question by collecting some basic properties, which all good conjectures have in common. He summarizes the following characteristics:

- *Simplicity*: The conjecture is short and easy to understand, containing basic concepts of the subject.
- *Surprise*: The conjecture establishes a surprising connection between seemingly unrelated concepts.
- *Generality*: The conjecture is true for a wide variety of objects, e.g. if speaking of graph theory, for a whole class of graphs.
- *Longevity*: Conjectures that have existed for a very long time, and fulfill all the properties mentioned, are usually the most interesting.
- *Fecundity*: Attempts to prove the conjecture have led to new concepts or new proof techniques [13].

If one think about leaving the process of conjecture-making to a machine, the points mentioned above are not very promising. First of all, we have to determine how a conjecture could be abstracted, so that a computer can process it. By knowing this problem, Fajtlowicz concluded that a program could easily produce conjectures of the form $I \leq J, I \leq J + K$ and $I + J \leq K + L$ with I, J, K and L being either an invariant of the mathematical object or a constant value. Hence, a conjecture can be considered as an unproven formula, which creates a relation between invariants, which is, generally speaking, considered as "surprising" in its form. However, this brings two further problems. First, how should one teach a program which statements are generally considered as "surprising"? Second, one can imagine that a program can produce an endless stream of statements of the form given above. How can we limit the produced output, or, more generally speaking, how can we automatically decide which statements are significant and which ones are not? In order to solve the second problem, Fajtlowicz invented a heuristic for deciding which conjectures are more significant than others. It is known as the "Fajtlowicz-Dalmatian Heuristic" and is implemented in his conjecture-making program *graffiti* and can be expressed with the following principle: "make the strongest conjecture for which no counterexample is known". The conjectures of *graffiti* are published in a series of articles by mainly focussing graph theory. The reader is referred to [34, 35, 36] and [37].

Larson summarizes the results and research attempts for the subject of automated conjecture-making in [68]. Then, together with Van Cleemput, he implemented a conjecture-making program, called *conjecturing*. They integrated the Fajtlowicz-Dalmatian heuristic and published a variety of articles covering their research in the field of automated conjecture-making. A first overview and some first results, mainly in the research field of graph theory, is given in [69]. In [70] they introduce the topic of automated conjecture-making for property-based conjectures. Usually, these conjectures have the form of an "If ... then ..."-statement. Since both authors have a background in chemical graph theory, they also participated in this research field. Together with Hutchinson et al. they investigated bounds for the domination number of benzenoids, which are graphs representing the carbon structure of molecules, defined by a closed path in the hexagonal lattice [83]. Muncy initially started the research for this topic under the supervision of Larson.

In the following, we are going to present the "Fajtlowicz-Dalmatian Heuristic" and then give an introduction to the program *conjecturing*. Also, we are going to give a guide on how to setup *conjecturing* in our *sage* environment. Finally, we are going to process the generated optimal 1-planar graphs of our generation algorithm to *conjecturing* and present some of the conjectures, which were made by the program.

6.1 Fajtlowicz-Dalmatian Heuristic

In the following, we are going to present the Fajtlowicz-Dalmatian heuristic, as the heart of the program *grafitti*, and as it is reimplemented in the program *conjecturing*.

Suppose some computable invariants $\alpha_1, \alpha_2, \cdots, \alpha_r$ of a mathematical object O of interest are given. Speaking in the terms of definition 2.50, each α_i is a function, which computes the desired invariant of O, hence $\alpha_i(O) = \alpha_i$. Additionally, we define some operations $f_1, f_2, \cdots, f_s$ of some algebraic system, which might be any binary or n-ary operator, e.g. "plus" or "times" (cf. [68]). For instance, a *term* is given by $f(\alpha_1, \alpha_2)$, which represents a new invariant. Then, "if t and s are terms, the expression $t \leq s$" is a *statement*, and therefore "for every object O, $t(O) \leq s(O)$, is a candidate for a conjecture" [68].

Example 6.1. Given a simple graph $G = (V, E)$, whereby $\Delta(G)$ is the maximum degree of G, $|V|$ is the number of vertices of G, and let $\gamma(G)$ be the domination number of G. Speaking in the language given above, G is the mathematical object O of interest, and $\Delta(G) = \alpha_1, \gamma(G) = \alpha_2$ and $|V| = \alpha_3$ are computable invariants of G.

Let f_m be the "minus"-operator. A candidate for a conjecture would be the following statement.

$$\alpha_2 \leq f_m(\alpha_3, \alpha_1) \tag{99}$$

$$\gamma(G) \leq |V| - \Delta(G) \tag{100}$$

One can imagine, that a program would produce an endless stream of statements of the form given above. The Fajtlowicz-Dalmatian heuristic consists of two steps for deciding if a new conjecture is significant. Suppose, we have a set $\Theta = \{O_1, \cdots, O_m\}$ of objects, which are in the same class as the object O, for which the program is calculating new conjectures. Additionally, assume that we have a set $\zeta = \{u_1, \cdots, u_l\}$ of pre-existing conjectures for the invariant t of interest of similar form, $t \leq u_1, t \leq u_2, \cdots, t \leq u_l$. Both ζ and Θ might be initially empty. Now, if $t \leq s$ is a new candidate for a conjecture, made by the conjecture-making program, the Fajtlowicz-Dalmatian heuristic consists of the following two steps (cf. [69]).

1. *Truth test*: The candidate conjecture $t \leq s$ is true for all $O \in \Theta$.

2. *Significance test*: There is at least one object $O \in \Theta$, for which $s(O) < u(O),\ \forall u \in \zeta$.

If a conjecture passes the two tests stated above, the conjecture is counted as significant. Notice, that Θ is usually generated with a graph generation program.
Since most invariants of interest are very hard to calculate, e.g. the domination number or the independence set number, Bushaw et al. has launched a project, named "House of Graphs", which is a database containing graphs of a variety of different graph classes, whereby most invariants of interest are pre-calculated. The graphs are persisted in the "graph6"-format. Unfortunately, we are not going to make advantage of this, since we are more interested in a proof of concept at this stage.

6.2 The conjecture-making program *conjecturing*

In the following we are going to present the program *conjecturing*. First, we are going to present general aspects of the conjecture-making process by specifying the main algorithm. Second, we are going to install the program in our *sage* environment.

Larson and Van Cleemput have experienced, during the development of the program *conjecturing*, that the number of produced conjectures is still unmanageable. To address this problem, they implemented the following rule in addition to the Fajtlowicz-Dalmatian heuristic: no more conjectures are generated than the number of objects in Θ.

The program *conjecturing* is implemented in the *C* programming language with a python binding for installing it as a *sage*-package. It is an expression generating program, whereby an expression is a rooted, labeled binary tree, "where each node has at most two children and each node with, respectively, two, one or no children is labeled with, respectively, a binary operator, an unary operator or an invariant" [69].

Example 6.2. Given the statement $\alpha \leq a + b$. The generated tree reads as follows:

- Two children $\equiv$ binary operator.
- One child $\equiv$ unary operator.
- No child $\equiv$ invariant.

Hence, the tree for the above statement is given by the one visualized in figure 6.1. Notice, that α, a and b do not have any children, therefore, they are invariants. $\leq$ and $+$ have two children, therefore, they are binary operators.

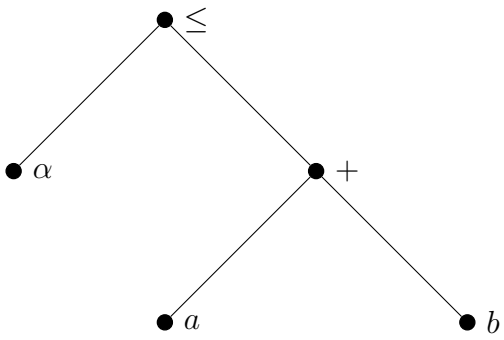


Figure 6.1: Rooted, labeled binary tree, generated by *conjecturing*.

In the following, we are going to present the main part of the algorithm of the program *conjecturing*. It consists of 6 steps as given in [69].

1. *Produce a finite stream of inequalities* of the form $t \leq s$, whereby s and t are "evaluable functions of the invariants" [69].
2. *Initialize a set of objects* Θ with at least one object. For example, either a graph generator could be used directly, or some graphs, which are stored in a database, could be loaded. A combination of both methods is also conceivable.
3. *Generate a stream of conjectures.* Each generated conjecture must pass the truth and the significance test in respect to the initial set of inequalities and to the initial set of objects Θ. The program needs a stopping condition, for which we have two possibilities. First, for each object in Θ there is a conjecture, which gives the exact value of the invariant on the left side of the inequality.

This is the best case possible, since "there is no other conjecture that can make better predictions [for the invariant of interest]" [69]. The program has achieved "exact predictive power for all objects" [69]. The second option is a hard-coded stopping condition, such as a specific time, set by the user.

4. *Remove insignificant conjectures.* If a conjecture is added to the set of conjectures, it is checked if the newly added conjecture makes any previously made conjecture insignificant. All insignificant conjectures are removed.

5. "*Search for a counterexample to any of these conjectures*" [69], e.g. by using a graph generator. For example, we could start making conjectures for relatively small optimal 1-planar graphs (8-11 vertices) and then search for counterexamples using larger optimal 1-planar graphs.

6. *If a counterexample is found, add it and repeat the conjecture generating process.* "The program can never make more conjectures than the number of objects it has stored: the reason is exactly because each conjecture in the conjectures store must give a better bound for at least one stored object than any other conjecture does" [69].

6.3 The setup of conjecturing in sage

In the following, we are going to setup the program *conjecturing* in our sage environment. We presuppose, that *conjecturing* is downloaded[25].

```
1 tar jxf conjecturing-0.12.tar.gz
```

Listing 23: Extracting the conjecturing package.

In listing 23 it is shown how to extract the downloaded archive. Next, we need to execute the following steps, in order to install conjecturing in our sage environment[26].

1. Copy the directory "spkg/conjecturing" into the directory SAGE_ROOT/build/pkgs, whereby SAGE_ROOT is the root directory of our local sage installation.

2. Copy the file "conjecturing-0.12.tar.gz" into SAGE_ROOT/upstream.

3. Navigate in SAGE_ROOT via the command line and run the following command:

```
1 ./sage --package fix-checksum conjecturing
```

Listing 24: Fixing the checksum of the conjecturing package.

[25] https://nvcleemp.github.io/conjecturing/ (visited on 01/13/2020).
[26] Step 1-4 are taken from the official readme-file.

4. Install the package using the following command:

```
./sage -i conjecturing
```

Listing 25: Installing conjecturing in sage.

5. In order to make *conjecturing* available in the sage notebook, one needs to copy the file "conjecturing.py" (located in conjecturing-0.12/sage of the extracted archive) into "~/.sage/sage_notebook.sagenb/home/admin/0/data".

Next, we need to load the python file in our sage notebook by executing the following command. The variable "DATA" is an internal system variable of the sage notebook, where the path to the directory is saved to where we copied the conjecturing python file.

```
load(DATA+'conjecturing.py')
```

Listing 26: Loading conjecturing.py in the sage notebook.

Now, we can test the installed *conjecturing* package by using an example, which is given in [104]. The code is shown in listing 27.

```
1 def max_degree(g):
2     return max(g.degree())
3
4 invariants = [Graph.size, Graph.order, max_degree]
5 objects = [graphs.CompleteGraph(n) for n in [3,4,5]]
6 conjecture(objects, invariants, 0)
```

Listing 27: Example: using conjecturing in sage.

The method *conjecture* takes three parameters as an input (l.6). The first parameter is a collection of mathematical objects, which consists of graphs in our case (l.5). The second parameter is a collection of invariants of the given objects, e.g. the maximum degree, the number of vertices (given by Graph.order) or the number of edges (given by Graph.size). We define a method for the maximum degree, called *max_degree* (l.1). The last parameter is the index, representing the invariant at that index of the provided collection of invariants, which is meant to be on the left hand side of the inequality of our conjectures. Listing 27 produces the following collection of conjectures.

```
1 size(x) <= 2*order(x)
2 size(x) <= max_degree(x)^2 - 1
3 size(x) <= 1/2*max_degree(x)*order(x)
```

Listing 28: Example: output of conjecturing.

6.4 Implementation of the conjecture-making process

In the following, we are going to implement the conjecture-making process for the optimal 1-planar graphs. First, we define the methods for calculating the invariants of interest We choose the following invariants for an optimal 1-planar graph $G = (V, E)$ with n vertices:

(i) The order $|V| = n$ of G.

(ii) The number of edges $|E|$ of G.

(iii) The number of crossing edges of G, which is exactly equal to $2(n-2)$, since the skeleton of G has $n-2$ faces and we add 2 crossing diagonals in each face in our generation algorithm (compare with section 3.2 and section 5).

(iv) The minimum degree $\delta(G) := min\{d(v) \mid v \in V\}$.

(v) The maximum degree $\Delta(G) := max\{d(v) \mid v \in V\}$.

(vi) The number of faces of the planar skeleton of G, given by $n-2$.

(vii) The diameter of G.

We are going to make conjectures for the upper and lower bounds of the independence number and the domination number.

In the following, we are first going to present the methods for calculating the invariants (i)-(vi). The method *max_degree* in listing 29 takes a graph object as an input, and returns the maximum degree of the graph. The method *degree* of a graph object in sage returns a list containing all the vertex degrees of the calling graph object. The method *max* takes a list as an input and returns its maximum value.

```
1 def max_degree(g):
2     return max(g.degree())
```

Listing 29: Calculating the maximum degree.

The minimum degree is calculated analogously by using the *min* method. The code is shown in listing 30.

```
1 def min_degree(g):
2     return min(g.degree())
```

Listing 30: Calculating the minimum degree.

We calculate the number of crossing edges of an optimal 1-planar graph G with n vertices with the formula $2(n-2)$, since the skeleton of G is a simple quadrangulation with $n-2$ faces and the two crossing diagonals are added in each face during the generation (see section 3.2). The implementation is shown in listing 31. The method takes a graph as an input and uses its *order* method (cf. [98]), which returns the number of vertices of the graph, in order to calculate the number of crossing edges.

```
1 def crossing_number(g):
2     return 2*(g.order() - 2)
```

Listing 31: Calculating the crossing number for optimal 1-planar graphs.

The number of faces of the quadrangular skeleton of an optimal 1-planar graph is calculated similarly. The implementation is shown in listing 32.

```
1 def num_skel_f(g):
2     return g.order() - 2
```

Listing 32: Calculating the number of faces of the skeleton of an optimal 1-planar graph.

The method *dominating_set* of a given graph returns its minimum dominating set as a collection of the involved vertices. We use the *len* function, which returns the size of a given collection, in order to calculate the domination number. The method is shown in listing 33.

```
1 def domination_number(g):
2     return len(g.dominating_set())
```

Listing 33: Calculating the domination number.

In listing 34 the method for calculating the independence number of a given graph is shown. First, it calculates all maximal independent sets of a given graph (l.3), using the *IndependentSet* class of the *independent_sets* module. The parameter *maximal* is set to *True*. As a result, only maximal independent sets are calculated. Then, we iterate over all maximal independent sets and return the maximum size (ll.5-8).

```
1 from sage.graphs.independent_sets import IndependentSets
2 def independence_number(g):
3     maximalIndependentSets = IndependentSets(g, maximal=True)
4     max = 0
5     for x in maximalIndependentSets:
6         size_x = len(x)
7         if max < size_x:
8             max = size_x
9     return max
```

Listing 34: Calculating the independence number.

Lastly, the method for making the conjectures for optimal 1-planar graphs is shown in listing 35. First, a collection, named *objects*, is initialized (l.1). Then, all optimal 1-planar graphs in the desired order range are generated using the *generateOptimal1PlanarGraphs*, which we have developed in section 5.2 (l.5-8). The generated graphs are stored in the collection *objects*. At this stage, we limit the generation of the optimal 1-planar graphs to 22 vertices. Next, we create a collection containing all the invariants of interest, as presented at the beginning of this section (ll.13-16). We store the index of the main invariant of interest, in our case either the domination number or the independence number, in the variable *mainInvariant* (l.18). Then, we establish the operators, which we want the algorithm to use on the right hand side of the inequalities[27]. They are as follows:

- -1: Substracts 1 from an invariant.
- $+1$: Adds 1 to an invariant.
- $*2$: An invariant is multiplied by 2.
- $/2$: An invariant is divided by 2.
- ^2: An invariant is raised to the power of 2.
- $-()$: The content, which can either be a single invariant or the result of a formula of invariants, of the brackets is negated.

[27]Notice, that the combination of several invariants by an operator is again an invariant.

- 1/: $\frac{1}{x}$, whereby x is an invariant.
- *sqrt*: Takes the square root of an invariant.

```
1  objects = []
2
3  # generate optimal 1 planar graphs
4  o1p = []
5  for n in range(8,23):
6      o1p = generateOptimal1PlanarGraphs(n)
7      for g in o1p:
8          objects.append(g)
9
10 # exchange domination_number with independence_number
11 # in order to make conjectures for it
12
13 invariants = [domination_number, Graph.size, Graph.order,
14         max_degree, min_degree, crossing_number,
15         num_skel_f, Graph.diameter]
16
17 mainInvariant = invariants.index(domination_number)
18
19 operators = { '-1', '+1', '*2', '/2', '^2', '-()', '1/',
20         'sqrt', 'ln', 'log10', '+', '*', 'max',
21         'min', '-', '/', '^'}
22
23 conjectures = conjecture(objects, invariants, mainInvariant,
24         upperBound=True, verbose=False, operators=operators)
25 print(conjectures)
```

Listing 35: Making conjectures for optimal 1-planar graphs using conjecturing in sage.

- *ln*: Takes the natural logarithm of an invariant.
- *log*10: Takes the common logarithm, which is the logarithm with base 10, of an invariant.
- +: Addition of two invariants.
- −: Substraction of two invariants
- ∗: Multiplication of two invariants.

- /: Division of two invariants.
- *max*: Returns the maximum of two invariants.
- *min*: Returns the minimum of two invariants.
- ^: An invariant is raised to the power of another invariant.

Next, we call the method *conjecture*, which is a python binding for the conjecture-making method, implemented in the *C* programming language. The method takes the objects, for which we want to make conjectures, the invariants of interest, the main invariant, which is the invariant on the left-hand side of the inequalities, and a collection of operators as an input. Furthermore, we pass two boolean arguments, whereby the argument *upperBound* is set to *True*, if we want to make conjectures for the upper bound of the main invariant. If *upperBound* is set to *False*, the method makes conjectures for the lower bound. Setting the argument *verbose* to *True*, sets the debugging level to *verbose*. We only use this, if further debugging output is needed, e.g. in case of strange behavior of the conjecture-making process.

6.5 Conjectures for optimal 1-planar graphs

We performed multiple runs for the upper and lower bounds of the domination number and the independence number. The results are shown in tables 6.1, 6.2, 6.3 and 6.4, whereby we will not discuss the meaningfulness of the delivered conjectures, since we are only interested in a proof of concept at this point.

1.	domination_number(x)	<=	((crossing_number(x)) / (order(x))) ^ 2
2.	domination_number(x)	<=	exp(log10(num_skel_f(x)))
3.	domination_number(x)	<=	(size(x)) / (crossing_number(x))
4.	domination_number(x)	<=	size(x)/num_skel_f(x)
5.	domination_number(x)	<=	num_skel_f(x)/min_degree(x) + 1
6.	domination_number(x)	<=	1/2*max_degree(x)
7.	domination_number(x)	<=	-max_degree(x) + order(x)
8.	domination_number(x)	<=	crossing_number(x)/max_degree(x) + 1
9.	domination_number(x)	<=	diameter(x)
10.	domination_number(x)	<=	order(x)/diameter(x) - 1

Table 6.1: Upper bounds for the domination number of optimal 1-planar graphs.

1.	domination_number(x)	>=	size(x)/crossing_number(x)
2.	domination_number(x)	>=	num_skel_f(x)/max_degree(x)
3.	domination_number(x)	>=	-2*max_degree(x) + num_skel_f(x)

Table 6.2: Lower bounds for the domination number of optimal 1-planar graphs.

1.	independence_number(x)	<=	ceil(arcsinh(log(size(x))))
2.	independence_number(x)	<=	1/2*min_degree(x)
3.	independence_number(x)	<=	-min_degree(x) + order(x)
4.	independence_number(x)	<=	floor(arcsinh(order(x)))
5.	independence_number(x)	<=	-minimum(num_skel_f(x), 1/4*min_degree(x)^2) + order(x)
6.	independence_number(x)	<=	crossing_number(x)/min_degree(x)
7.	independence_number(x)	<=	-floor(tan(size(x))) + num_skel_f(x)
8.	independence_number(x)	<=	ceil(log(crossing_number(x)))
9.	independence_number(x)	<=	floor(crossing_number(x)/min_degree(x))
10.	independence_number(x)	<=	2*(min_degree(x) - num_skel_f(x) + 1)^2
11.	independence_number(x)	<=	-size(x)/(max_degree(x) - order(x))
12.	independence_number(x)	<=	sqrt(max_degree(x) + order(x) - 1)
13.	independence_number(x)	<=	-1/4*max_degree(x) + 1/2*order(x)
14.	independence_number(x)	<=	-min_degree(x) + order(x)
15.	independence_number(x)	<=	-diameter(x) + 1/2*order(x)

Table 6.3: Upper bounds for the independence number of optimal 1-planar graphs.

1.	independence_number(x)	>=	size(x)/crossing_number(x)
2.	independence_number(x)	>=	1/4*num_skel_f(x) - 1/4
3.	independence_number(x)	>=	sqrt(2*max_degree(x) - 2*min_degree(x))
4.	independence_number(x)	>=	-max_degree(x)^2 + size(x)
5.	independence_number(x)	>=	1/2*max_degree(x) - 1/2*min_degree(x)
6.	independence_number(x)	>=	log(1/2*max_degree(x))^2
7.	independence_number(x)	>=	crossing_number(x)/(max_degree(x) + 1)
8.	independence_number(x)	>=	num_skel_f(x)/(min_degree(x) - 1)

Table 6.4: Lower bounds for the independence number of optimal 1-planar graphs.

At this point, there are many possible ways to improve the conjecture-making process. First, we could include already known bounds for the invariant of interest, e.g. by using the database of the "House of Graphs" project. We could also take a closer look at the graphs on which we carry out the conjectures and then optimize the search for counter examples by using a larger set of graphs.

7 Conclusions

In this thesis we have essentially pursued two goals: first, the development and implementation of an algorithm for the generation of optimal 1-planar graphs, and second, using the generated graphs as an input for a conjecture-making program. We started by investigating the needed principles of graph theory, whereby we have focused the graph isomorphism problem and the general topology of planar graphs.

Two graphs are called isomorphic, if there exists an adjacency preserving bijection between their two vertex sets. We have shown that graph isomorphism is an equivalence relation, and therefore a natural division of the graphs of a given set of graphs into equivalence classes is induced. This relationship in particular is very important for the generation algorithm, since it lays the foundation for isomorph rejection, which describes the process of rejecting generated graphs of an already created equivalence class. The best known algorithm for detecting isomorphic graphs has been developed by McKay and Piperno, whereby the isomorph rejection is based on canoncial labeling. A canonical labeling is a complete graph invariant, and two given isomorphic graphs become identical when they are canonically labeled [76]. The algorithms behind that process are implemented in the programs *nauty* and *Traces* [78, 87].

Then, we have considered the topology of planar graphs. We started by giving a short introduction to basic terms of general topology, such as *topological spaces* or *homeomorphism*. We have seen that an edge of an embedded graph is an (polygonal) arc, for which we have shown homeomorphism to the unit interval, whereby its endpoints are the vertices of the graph. We used these investigations to give a definition of a planar embedding, which is generally known as a *plane graph*, and showed that the *faces* of a plane graph G, are its *regions*. The regions are open subsets of $\mathbb{R}^2$, defined as $\mathbb{R}^2 \setminus G$. We have given an example in order to show that exactly one of the regions is unbounded. This unbounded region is called the *outer face* of G. Since most literature introduces the terms *planarity* and 1-*planarity* with basic terms of topology, we considered this introduction a necessity. Furthermore, we believe that the generation of the optimal 1-planar graphs or of its skeletons could be more easily understood.

In order to present an embedding in a machine-readable form, we considered a representation of plane graphs as combinatorial objects called *rotation systems*. A rotation system is a set of cyclic permutations, whereby each permutation describes the (clockwise or anti-clockwise) ordering of the edges incident with a vertex. We call a permutation π_v the local rotation of the edges at v. We have seen, that two embeddings can be considered identical if their respective rotation systems are the same. Using these definitions, we defined two isomorphisms which take the actual graph embeddings into account. The first one is called O-P isomorphism, which, in addition to abstract graph isomorphism, requires an edge bijection that preserves the cyclic order of the edges.

The second one is called O-R isomorphism, which, in addition to abstract graph isomorphism, requires an edge bijection that reverses the cyclic order of the edges. We have shown, that O-P isomorphism is an equivalence relation, whereby O-R isomorphism is not. Both isomorphisms are implemented in the program *plantri* by Brinkmann and McKay for isomorph rejection and therefore are important for the generation of optimal 1-planar graphs.

After introducing 1-planar graphs, we investigated the structure and properties of optimal 1-planar graphs. We have shown, that a 1-planar graph can have a maximum of $4n - 8$ edges and that they are obtained from the simple quadrangulations by adding the crossing diagonal edges in each of the faces. A simple quadrangulation is a simple plane graph, for which every face is bounded by a 4-cycle. We have proven, that the average degree of a simple quadrangulation Q is less then 4. Using this investigation, we showed that the connectivity of Q cannot be more than 3. Then, we proved that optimal 1-planar graphs with n vertices only exist for $n = 8$ and $n \geq 10$. The proofs have already been given or indicated by Bodendiek et al., but were presented in more detail.

Then, we have introduced the topic of recursive graph generation. In general, a recursive generator for a family of graphs consists of a set of expansion rules and a set of irreducible members of the given family. Using the expansion rules, the irreducible members are expanded to larger graphs, while remaining inside its graph family. We have presented a generator for the simple triangulations.

Next, we considered the generation of simple quadrangulations as given by Brinkmann et al. We presented the expansion rules and gave several examples. We were not so much concerned with theorems and formal proofs in this section, but much more with the basic idea of the generation, since it is strongly related to the generation of the optimal 1-planar graphs.

The generation of the simple quadrangulations is implemented in the prorgram *plantri* by Brinkmann and McKay. We presented plantri and showed the setup on a linux machine. Furthermore, we presented the different output formats. Second, we have set up plantri in a *SageMath* environment.

We did a runtime performance comparison between the "plain" version of plantri and the one integrated in sage. We generated the simple 3-connected quadrangulations starting with 8 vertices up to 30 and measured the execution times. The "plain" version of plantri drastically outperformed the version integrated in sage. However, we have not examined the exact causes in more detail. Possible reasons could be the parsing of the planar code in sage internal graph objects. An overhead in the way sage calls external code could also be conceivable.

Next, we discussed four possibilities for the implemenation of an algorithm for the generation of the optimal 1-planar graphs. We decided to use the integrated version of plantri in sage to generate the skeletons of the optimal 1-planar graphs and then insert

the intersecting diagonals in each face of the planar skeletons. Since we wanted to use the generated graphs for automated conjecture-making, we were not so much concerned with runtime performance. Additionally, the subsequent processing to the conjecture-making program was made much easier. We have formalized our approach by presenting the expansion rules for the recursive generation of the optimal 1-planar graphs as given by Suzuki. We have shown the similarities to the generation of the simple quadrangulations as given by Brinkmann et al.

We implemented the generation algorithm for optimal 1-planar graphs in sage and did a runtime performance test for our implementation as well. Our time measurements provided data points for graphs of order 8 up to 25. In order to get an idea of the runtime for larger graphs, we performed a regression analysis. Our model reaches a RMSE of approximately 5.22 with the given data ranging from 0.00 to 1455.69, which can be considered as accurate. Using the calculated model, we forecasted the runtime of our implementation for generating all optimal 1-planar graphs with 30 vertices, which resulted in a predicted runtime of approximately 918724 seconds, which is not very efficient in comparison to the generation of its skeletons. It is expected that implementing the generation of the optimal 1-planar graphs directly into plantri or implementing the expansion rules given by Suzuki will give much better results. For this reason, implementing the generation algorithm while keeping runtime efficiency in mind and then comparing the results with our implementation would be interesting.

Finally, we introduced the topic of automated graph conjecture-making. First, we have presented some characteristics of "good" conjectures as collected by Bondy. Second, we differentiated between conjectures having the form of inequalities and so called property based conjectures. An inequality conjecture can be considered as an unproven formula, which establishes a relation between different graph invariants. A property based conjecture is usally an "If ... then ..."-statement. We have focused on inequality conjectures for the domination- and independence number. In order to limit the stream of conjectures made by a conjecture-making program, Fajtlowicz developed an heuristic, called the Fajtlowicz-Dalmatian heuristic. We presented the heuristic and a program, which implements this heuristic. This program is called *conjecturing* and was implemented by Larson and Van Cleemput. We presented *conjecturing* and set it up in our sage environment. Next, we implemented the conjecture-making process for making conjectures for the domination- and the independence number for optimal 1-planar graphs. We used some invariants of optimal 1-planar graphs, such as the number of crossing edges or the number of faces of its planar skeletons, as they are easy to calculate. Last but not least, we performed mutliple runs for the upper and lower bounds of the domination- and the independence number for optimal 1-planar graphs using *conjecturing*, whereby we have not discussed the meaningfulness of the delivered conjectures.

This thesis reveals possible further research topics. First of all, one could optimize the algorithm for the generation of the optimal 1-planar graphs and its implementation and do a comparison with our results. One could also try to generate other subfamilies of 1-planar graphs, e.g. outer 1-planar graphs, and use them for automated conjecture-making. Furthermore, one could also examine the conjectures, which were made for optimal 1-planar graphs, and prove or disprove their correctness. Additionally, one could integrate already known bounds for the significance test, which would most likely yield into better results.

It can also be determined that the conjecture-making process seems to be outdated considering todays knowledge of machine learning. For example, the conjecture-making process seems to be suitable for a genetic algorithm. In fact, Absil and Mélot implemented a conjecture-making process, which uses a genetic algorithm, in a program called *digenes*. A comparison between *digenes* and *conjecturing* could be interesting. Furthermore, modern machine learning techniques could be exploited for discovering new structural relationships between different graph invariants.

Speaking of machine learning, we quickly ask ourselves the following question: why should we limit ourselves to the automation of the conjecture-making process? Could it be possible to leave the subsequent proof of the conjectures to a machine as well? In fact, this idea is not new and is investigated in a research field called *computer assisted reasoning*. First approaches for automating mathematical reasoning were done by Trybulec and Blair in 1985 with a software system called the *Mizar* system. In 2010 a proof for euler's polyhedron formula was formalized by Alama using Mizar. Combining the research areas of automated conjecture-making and computer assisted reasoning could open up completely new opportunities.

References

[1] Romain Absil and Hadrien Mélot. "Digenes: genetic algorithms to discover conjectures about directed and undirected graphs". In: *arXiv preprint arXiv:1304.7993* (2013).

[2] Jesse Alama. "Euler's Polyhedron Formula in mizar". In: *International Congress on Mathematical Software*. Springer. 2010, pp. 144–147.

[3] Christopher Auer et al. "1-Planarity of Graphs with a Rotation System." In: *J. Graph Algorithms Appl.* 19.1 (2015), pp. 67–86.

[4] Christopher Auer et al. "Outer 1-planar graphs". In: *Algorithmica* 74.4 (2016), pp. 1293–1320.

[5] Christopher Auer et al. "Recognizing outer 1-planar graphs in linear time". In: *International Symposium on Graph Drawing*. Springer. 2013, pp. 107–118.

[6] Eric Ayeh. "An Investigation Into Graph Isomorphism Based Zero-knowledge Proofs". PhD thesis. Citeseer, 2009.

[7] László Babai. "Graph isomorphism in quasipolynomial time". In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. ACM. 2016, pp. 684–697.

[8] Armen Bagdasaryan. "Some Euler-type formulas for planar graphs". In: *International Journal of Pure and Applied Mathematics* 79 (Oct. 2012).

[9] David Barnette. "On generating planar graphs". In: *Discrete Mathematics* 7.3-4 (1974), pp. 199–208.

[10] Vladimir Batagelj. "An inductive definition of the class of 3-connected quadrangulations of the plane". In: *Discrete Mathematics* 78.1 (1989), pp. 45 –53.

[11] V. Berhard. *Zur Morphologie der Polyeder*. Teubner, 1891.

[12] R. Bodendiek, H. Schumacher, and K. Wagner. "Über 1-optimale Graphen". In: *Mathematische Nachrichten* 117.1 (1984), pp. 323–339.

[13] Adrian Bondy. "Beautiful conjectures in graph theory." In: *Eur. J. Comb.* 37 (2014), pp. 4–23.

[14] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*. Vol. 290. Macmillan London, 1976.

[15] Nicolas Bourbaki. *Algebra I*. Addison-Wesley Publishing Company, 1974.

[16] Robert Bowen and Stephen Fisk. "Generation of triangulations of the sphere". In: *Mathematics of Computation* 21.98 (1967), pp. 250–252.

[17] Franz J. Brandenburg. "Recognizing optimal 1-planar graphs in linear time". In: *Algorithmica* 80.1 (2018), pp. 1–28.

[18] Glen E. Bredon. *Topology and geometry*. Vol. 139. Springer Science & Business Media, 2013.

[19] Gunnar Brinkmann and Brendan McKay. *Guide to using plantri (version 5.0)*. URL: https://users.cecs.anu.edu.au/~bdm/plantri/plantri-guide.txt (visited on 11/10/2019).

[20] Gunnar Brinkmann and Brendan McKay. *plantri and fullgen*. URL: http://users.cecs.anu.edu.au/~bdm/plantri/ (visited on 12/26/2019).

[21] Gunnar Brinkmann, Brendan D. McKay, et al. "Fast generation of planar graphs". In: *MATCH Commun. Math. Comput. Chem* 58.2 (2007), pp. 323–357.

[22] Gunnar Brinkmann et al. "Generation of simple quadrangulations of the sphere". In: *Discrete mathematics* 305.1-3 (2005), pp. 33–54.

[23] Gunnar Brinkmann et al. *House of Graphs: a database of interesting graphs*. URL: http://hog.grinvin.org (visited on 12/26/2019).

[24] Csilla Bujtás and Sandi Klavžar. "Improved upper bounds on the domination number of graphs with minimum degree at least five". In: *Graphs and Combinatorics* 32.2 (2016), pp. 511–519.

[25] N. Bushaw, Craig E. Larson, and Nicolas Van Cleemput. "Automated Conjecturing VII: The Graph Brain Project & Big Mathematics". In: *arXiv* (2018). eprint: 1801.01814.

[26] William A. Coppel. *Number Theory: An introduction to mathematics*. 2nd ed. Springer Science & Business Media, 2009.

[27] Brandon Curtis. *Sage Math Tutorial - Data Fitting*. URL: http://sage.brandoncurtis.com/data-fitting.html (visited on 01/10/2020).

[28] Ermelinda DeLaVina. "Some history of the development of Graffiti". In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 69 (2005), p. 81.

[29] Matt DeVos. *Lecture Notes - Planarity*. URL: http://www.sfu.ca/~mdevos/notes/graph/445_planarity.pdf (visited on 12/11/2019).

[30] Walter Didimo, Giuseppe Liotta, and Salvatore Romeo. "A Graph Drawing Application to Web Site Traffic Analysis". In: *Journal of Graph Algorithms Applications* 15 (Jan. 2011), pp. 229–251.

[31] Reinhard Diestel. private communication. Oct. 29, 2019.

[32] Reinhard Diestel. *Graph Theory (Graduate Texts in mathematics)*. 5th ed. Vol. 173. Springer, 2017.

[33] Igor Fabrici and Tomáš Madaras. "The structure of 1-planar graphs". In: *Discrete Mathematics* 307.7-8 (2007), pp. 854–865.

[34] Siemion Fajtlowicz. "On conjectures of Graffiti". In: *Annals of Discrete Mathematics*. Vol. 38. Elsevier, 1988, pp. 113–118.

[35] Siemion Fajtlowicz. "On conjectures of Graffiti, II". In: *Congressus Numerantium*. Vol. 60. 1987, pp. 189–197.

[36] Siemion Fajtlowicz. "On conjectures of Graffiti, III". In: *Congressus Numerantium*. Vol. 66. 1988, pp. 23–32.

[37] Siemion Fajtlowicz. "On conjectures of Graffiti, IV". In: *Congressus Numerantium*. Vol. 70. 1990, pp. 231–240.

[38] Siemion Fajtlowicz. "Toward fully automated fragments of graph theory". In: (2003).

[39] Gerd Fischer. *Lineare Algebra - Eine Einführung für Studienanfänger*. 18th ed. Springer Spektrum, 2013.

[40] Python Software Foundation. *The pass statement*. URL: https://docs.python.org/2.0/ref/pass.html (visited on 01/05/2020).

[41] Frank Gaitan and Lane Clark. "Graph isomorphism and adiabatic quantum computing". In: *Physical Review A* 89.2 (2014).

[42] Fănică Gavril. "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph". In: *SIAM Journal on Computing* 1.2 (1972), pp. 180–187.

[43] Sumanta Ghosh and Piyush P. Kurur. "Permutation groups and the graph isomorphism problem". In: *Perspectives in Computational Complexity*. Springer, 2014, pp. 183–202.

[44] Alexander Grigoriev and Hans L. Bodlaender. "Algorithms for graphs embeddable with few crossings per edge". In: *Algorithmica* 49.1 (2007), pp. 1–11.

[45] Jonathan L. Gross and Thomas W. Tucker. *Topological Graph Theory*. Wiley-Interscience, 1987.

[46] Jonathan L. Gross and Jay Yellen. *Graph theory and its applications*. CRC press, 2013.

[47] Pierre Hansen et al. "What Forms Do Interesting Conjectures Have in Graph Theory?" In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 69 (2005).

[48] Jorge (https://https://math.stackexchange.com/users/33907/jorge-fern%c3%a1ndez-hidalgo) Hidalgo. *G is connected,planar, simple graph with v vertices and e edges. Prove if every face is isomorphic to C_k then $e = \frac{k(v-2)}{k-2}$*. URL: https://math.stackexchange.com/questions/2121835/g-is-connected-planar-simple-graph-with-v-vertices-and-e-edges-prove-if?rq=1 (visited on 12/16/2019).

[49] Randall R. Holmes. *Abstract Algebra I*. Auburn University, Nov. 2018.

[50] Charles D. Homans. *On the group-theoretic properties of the automorphism groups of various graphs*. 2016.

[51] Seok-Hee Hong et al. "A linear-time algorithm for testing outer-1-planarity". In: *Algorithmica* 72.4 (2015), pp. 1033–1054.

[52] John E. Hopcroft and Jin-Kue Wong. "Linear time algorithm for isomorphism of planar graphs (preliminary report)". In: *Proceedings of the sixth annual ACM symposium on Theory of computing*. ACM. 1974, pp. 172–184.

[53] *How to pass parameters of a function when using timeit.Timer()*. URL: https://stackoverflow.com/questions/5086430/how-to-pass-parameters-of-a-function-when-using-timeit-timer (visited on 01/05/2020).

[54] Dávid Hudák, Tomáš Madaras, and Yusuke Suzuki. "On properties of maximal 1-planar graphs". In: *Discussiones Mathematicae Graph Theory* 32.4 (2012), pp. 737–747.

[55] Laura Hutchinson et al. "Automated conjecturing VI: Domination number of benzenoids". In: *MATCH Communications in Mathematical and in Computer Chemistry* 80.3 (2018), pp. 821–834.

[56] Viswanathan K. Iyer. *NP-Completeness of Independent Set*. URL: https://www.nitt.edu/home/academics/departments/cse/faculty/kvi/NPC%20INDEPENDENT%20SET-CLIQUE-VERTEX%20COVER.pdf (visited on 01/28/2019).

[57] Klaus Jänich. *Lineare Algebra*. Springer-Verlag, 2013.

[58] Gareth Jones and David Singerman. "Theory of Maps on Orientable Surfaces". In: *Proceedings of The London Mathematical Society - PROC LONDON MATH SOC* s3-37 (Sept. 1978), pp. 273–307.

[59] Mohammadreza Jooyandeh. "Recursive Algorithms for Generation of Planar Graphs". PhD thesis. The Australian National University, 2014.

[60] Michael Joswig et al. "Vertex-Facet Incidences of Unbounded Polyhedra". In: *Advances in Geometry* 1 (July 2000). DOI: 10.1515/advg.2001.002.

[61] Richard M. Karp. "Reducibility among combinatorial problems". In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[62] Dan M. Katz (https://math.stackexchange.com/users/6300/dan-m katz). *Meaning of topology and topological space*. URL: https://math.stackexchange.com/questions/137944/meaning-of-topology-and-topological-space (visited on 10/28/2019).

[63] Michael Kaufmann. "Graph Drawing Algorithms for Bioinformatics". In: *Informatik bewegt: Informatik 2002 - 32. Jahrestagung der Gesellschaft für Informatik e.v. (GI), 30. September - 3.Oktober 2002 in Dortmund, Ergänzungsband*. 2002, p. 65.

[64] Anthony W. Knapp. *Basic algebra*. Springer Science & Business Media, 2007.

[65] Vladimir P. Korzhik and Bojan Mohar. "Minimal Obstructions for 1-Immersions and Hardness of 1-Planarity Testing". In: *Lecture Notes in Computer Science* (2009), 302–312. ISSN: 1611-3349. DOI: 10.1007/978-3-642-00219-9_29. URL: http://dx.doi.org/10.1007/978-3-642-00219-9_29.

[66] Richard E. Ladner. "On the structure of polynomial time reducibility". In: *Journal of the ACM (JACM)* 22.1 (1975), pp. 155–171.

[67] Sergei K. Lando and Alexander K. Zvonkin. *Graphs on surfaces and their applications*. Vol. 141. Springer Science & Business Media, 2013.

[68] Craig E. Larson. "A survey of research in automated mathematical conjecture-making". In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 69 (2005), p. 297.

[69] Craig E. Larson and Nico Van Cleemput. "Automated conjecturing I: Fajtlowicz's Dalmatian heuristic revisited". In: *Artificial Intelligence* 231 (2016), pp. 17–38.

[70] Craig E. Larson and Nicolas Van Cleemput. "Automated conjecturing III - Property-relations conjectures". In: *Annals of Mathematics and Artificial Intelligence* 81.3-4 (2017), pp. 315–327.

[71] Ngoc C. Lê. "Algorithms for the Maximum Independent Set Problem". PhD thesis. University of Freiberg, 2014.

[72] Eric Leschinski. *Is there a library function for Root mean square error (RMSE) in python?* URL: https://stackoverflow.com/questions/17197492/is-there-a-library-function-for-root-mean-square-error-rmse-in-python/37861832#37861832 (visited on 01/10/2020).

[73] MathWorks. *Evaluating Goodness of Fit*. URL: https://de.mathworks.com/help/curvefit/evaluating-goodness-of-fit.html (visited on 01/10/2020).

[74] Brendan McKay. *graph formats*. URL: http://users.cecs.anu.edu.au/~bdm/data/formats.html (visited on 12/26/2019).

[75] Brendan D. McKay. "Isomorph-free exhaustive generation". In: *Journal of Algorithms* 26.2 (1998), pp. 306–324.

[76] Brendan D. McKay and Adolfo Piperno. *nauty and Traces*. URL: http://pallini.di.uniroma1.it/Introduction.html (visited on 01/20/2020).

[77] Brendan D. McKay and Adolfo Piperno. "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112.

[78] Brendan D McKay et al. *Practical graph isomorphism*. Department of Computer Science, Vanderbilt University Tennessee, USA, 1981.

[79] Robert Miller et al. *Common Graphs*. URL: http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/graph_generators.html#sage.graphs.graph_generators.GraphGenerators.quadrangulations (visited on 12/30/2019).

[80] Bojan Mohar and Carsten Thomassen. *Graphs on surfaces*. Vol. 2. Johns Hopkins University Press Baltimore, 2001.

[81] Cristopher Moore, Alexander Russell, and Piotr Śniady. "On the impossibility of a quantum sieve algorithm for graph isomorphism". In: *SIAM Journal on Computing* 39.6 (2010), pp. 2377–2396.

[82] Lee Mosher and Levi (https://math.stackexchange.com/users/513190/levi-ryffel) Ryffel. *Show there is a homeomorphism between a line segment and the unit interval*. URL: https://math.stackexchange.com/questions/3413688/show-there-is-a-homeomorphism-between-a-line-segment-and-the-unit-interval (visited on 10/29/2019).

[83] David Muncy. "Automated Conjecturing Approach for Benzenoids". Phd. Thesis. Virginia Commonwealth University, 2016.

[84] James Munkres. *Topology*. Pearson Education, 2014.

[85] Peter J. Olver and Peter John Olver. *Classical invariant theory*. Vol. 44. Cambridge University Press, 1999.

[86] Charles C. Pinter. *A Book of Set Theory*. New York: Dover Publications Inc., 2014.

[87] Adolfo Piperno. "Search Space Contraction in Canonical Labeling of Graphs (Preliminary Version)". In: *CoRR* (2008).

[88] José Luis López Presa. "Efficient algorithms for graph isomorphism testing". PhD thesis. Department of Computer Science, Universidad Rey Juan Carlos, 2009.

[89] Gerhard Ringel. "Ein sechsfarbenproblem auf der Kugel". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*. Vol. 29. 1. Springer. 1965, pp. 107–117.

[90] Luke Rodriguez. *Automorphism groups of simple graphs*. 2014.

[91] Keijo Ruohonen. *Graph Theory*. Tampere University of Technology, 2013.

[92] Pascal Schweitzer. "Problems of unknown complexity: graph isomorphism and Ramsey theoretic numbers". PhD thesis. Graph isomorphism and Ramsey theoretic numbers. Diss. University of Saarlandes, 2009.

[93] Rachna Somkunwar and Vinod Moreshwar Vaze. "A Comparative Study of Graph Isomorphism Applications". In: *International Journal of Computer Applications* 162.7 (2017).

[94] Saul Stahl and Catherine Stenson. *Introduction to topology and geometry*. 2nd ed. John Wiley and Sons, Inc., 2013.

[95] David Steiner. "Maximal Clique Variants". MA thesis. University of Glasgow, 2014.

[96] Wilson A. Sutherland. *Introduction to Metric and Topological Spaces*. 2nd ed. Oxford University Press, 2009.

[97] Yusuke Suzuki. "Optimal 1-planar graphs which triangulate other surfaces". In: *Discrete Mathematics* 310.1 (2010), pp. 6–11.

[98] The Sage Development Team. *Generic graphs (common to directed/undirected)*. URL: http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/generic_graph.html (visited on 01/13/2020).

[99] Etsuji Tomita and Hiroaki Nakanishi. "Polynomial-time solvability of the maximum clique problem". In: *Proc. of the European Computing Conference-ECC*. 2009, pp. 203–208.

[100] Glenna Toomey. *Algebraic Graph Theory: Automorphism Groups and Cayley graphs*. 2014.

[101] Jacobo Torán and Fabian Wagner. "The Complexity of Planar Graph Isomorphism". In: (2010).

[102] William F. Trench. *Introduction to real analysis*. Pearson Education, 2013.

[103] Andrzej Trybulec and Howard A Blair. "Computer assisted reasoning with MIZAR." In: *IJCAI*. Vol. 85. 1985, pp. 26–28.

[104] Nicolas Van Cleemput. *Conjecturing for Sage*. URL: https://nvcleemp.github.io/conjecturing/ (visited on 01/13/2020).

[105] Eric W. Weisstein. *Invariant*. URL: http://mathworld.wolfram.com/Invariant.html (visited on 01/06/2020).

[106] Eric W. Weisstein. *Quadrilateral*. URL: http://mathworld.wolfram.com/Quadrilateral.html (visited on 12/09/2019).

[107] Dainis Zeps and Paulis Ķikusts. "How to draw combinatorial maps?" In: (2013).

A Appendix

A.1 Definitions

Definition A.1. [84, p.17](*Composite*) Given two functions $f : A \to B$ and $g : B \to C$, we define the *composite* $g \circ f$ of f and g as the function $g\ circ f : A \to C$ defined by the equation $(g \circ f)(a) = g(f(a))$. Formally, $g \circ f : A \to C$ is the function whose rule is

$$\{(a, c) \mid For\ some\ b \in B, f(a) = b\ and\ g(b) = c\}. \tag{101}$$

Definition A.2. [84, p.18](*Injective function*) A function $f : A \to B$ is said to be *injective* (or *one-to-one*) if for each pair distinct points of A, their images under f are distinct. More formally we write: f is injective if

$$[f(a) = f(a')] \Rightarrow [a = a'] \tag{102}$$

Definition A.3. [84, p.18](*Surjective function*) A function $f : A \to B$ is said to be *surjective* (or f is said to map A *onto* B) if every element of B is the image of some element of A under the function f. More formally we write: f is surjective if

$$[b \in B] \Rightarrow [b = f(a)\ for\ at\ least\ one\ a \in A] \tag{103}$$

Definition A.4. [84, p.18](*Bijective function*) A function $f : A \to B$ is said to be *bijective* (or is called a *one-to-one correspondence*) if f is both injective and surjective.

Definition A.5. [84, p.18](*Inverse*) If a function f is bijective, there exists a function from B to A called the *inverse* of f. It is denoted by f^{-1} and is defined by letting $f^{-1}(b)$ be that unique element of $a \in A$ for which $f(a) = b$.

Theorem A.1. [86] If a $f : A \to B$ is a bijective function, then $f^{-1} : B \to A$ is a bijective function[28].

Theorem A.2. [86] Suppose $f : A \to B, g : B \to C$ and $g \circ f : A \to C$ are functions.

(i) If f and g are injective, then $g \circ f$ is injective.

(ii) If f and g are surjective, then $g \circ f$ is surjective.

(iii) If f and g are bijective, then $g \circ f$ is bijective.[29]

[28] See [86] for the proof.
[29] See [86] for the proof.

Definition A.6. [84, p.21](*Relation*) A *relation* on a set A is a subset R of the cartesian product $A \times A$. If R is a relation on A, we use the notation xRy to mean the same thing as $(x, y) \in R$. We read it "x is in the relation R to y".

Definition A.7. [84, p.22](*Equivalence Relation*) An *equivalence relation* on a set A is a relation $\sim$ on A haven the following three properties:

1. (*Reflexivity*) $x \sim x$ for every x in A.
2. (*Symmetry*) If $x \sim y$, then $y \sim x$.
3. (*Transitivity*) If $x \sim y$ and $y \sim z$, then $x \sim z$.

Definition A.8. [84, p.22](*Equivalence class*) Given an equivalence relation $\sim$ on a set A and an element x of A, we define a certain subset E of A, called the *equivalence class* determined by x, by the equation

$$E = \{y \mid y \in A \text{ and } y \sim x\} \tag{104}$$

Note that the equivalence class E determined by x contains x, since $x \sim x$.

Definition A.9. [90, p.1](*Group*) A *group* is a set S together with an operation $*$ such that:

(i) $\forall a, b \in S$, the element $a * b = c$ has the property that $c \in S$.

(ii) $\forall a, b \in S$, the equality $a * (b * c) = (a * b) * c$ holds.

(iii) There exists an element $e \in S$ such that $a * e = e * a = a,\ \forall a \in S$.

(iv) $\forall a \in S$ there exists an element $b \in S$ such that $a * b = b * a = e$.

Definition A.10. [64, p.119](*Abelian group*) An *abelian group* is a group G with the additional property

$$a * b = b * a,\ \forall a, b \in G \text{ (commutative law)} \tag{105}$$

Definition A.11. (Cf. [15, p.30])(*Subgroup*) Let S be a group with an operation $*$. A *subgroup* of S is a subset H of S such that H satisfies the properties of a group under the operation $*$. We write $H \subseteq G$. If $H \subset G$, we call H a *proper* subgroup of G.

A.2 Proofs

Lemma A.1. Given a permutation ϕ and its inverse ϕ^{-1}, then $\phi \circ \phi^{-1} = \phi^{-1} \circ \phi = id$.

Proof. First, we show that $\phi \circ \phi^{-1} = id$. Second, we show that $\phi^{-1} \circ \phi = id$.

(i) The equation

$$\phi \circ \phi^{-1} = \begin{pmatrix} v_0 & v_1 & \cdots & v_n \\ \phi(v_0) & \phi(v_1) & \cdots & \phi(v_n) \end{pmatrix} \circ \begin{pmatrix} \phi(v_0) & \phi(v_1) & \cdots & \phi(v_n) \\ v_0 & v_1 & \cdots & v_n \end{pmatrix} \tag{106}$$

leads to the following mappings

$$\phi(v_0) \mapsto v_0 \mapsto \phi(v_0) = \phi(v_0) \mapsto \phi(v_0) \tag{107}$$
$$\phi(v_1) \mapsto v_1 \mapsto \phi(v_1) = \phi(v_1) \mapsto \phi(v_1) \tag{108}$$
$$\vdots \tag{109}$$
$$\phi(v_n) \mapsto v_n \mapsto \phi(v_n) = \phi(v_n) \mapsto \phi(v_n) \tag{110}$$

The above mappings obviously represent the identity permutation, thus $\phi \circ \phi^{-1} = id$.

(ii) The equation

$$\phi^{-1} \circ \phi = \begin{pmatrix} \phi(v_0) & \phi(v_1) & \cdots & \phi(v_n) \\ v_0 & v_1 & \cdots & v_n \end{pmatrix} \circ \begin{pmatrix} v_0 & v_1 & \cdots & v_n \\ \phi(v_0) & \phi(v_1) & \cdots & \phi(v_n) \end{pmatrix} \tag{111}$$

leads to the following mappings

$$v_0 \mapsto \phi(v_0) \mapsto v_0 = v_0 \mapsto v_0 \tag{112}$$
$$v_1 \mapsto \phi(v_1) \mapsto v_1 = v_1 \mapsto v_1 \tag{113}$$
$$\vdots \tag{114}$$
$$v_n \mapsto \phi(v_n) \mapsto v_n = v_n \mapsto v_n \tag{115}$$

The above mappings obviously represent the identity permutation, thus $\phi^{-1} \circ \phi = id$.

Therefore, $\phi \circ \phi^{-1} = \phi^{-1} \circ \phi = id$. □

Lemma A.2. Given $x, y, z \in \mathbb{Z}$ with $z = x + y$. Let x, y be odd, then z is even.

Proof. By definition: $x = 2m + 1,\ m \in \mathbb{Z}$ and $y = 2n + 1, \in \mathbb{Z}$ (cf. [26]).

$$\Rightarrow\ 2m + 1 + 2n + 1 = z \tag{116}$$
$$\Leftrightarrow\ 2(m + n + 1) = z \tag{117}$$

Let $(m + n + 1) = z'$; clearly $z' \in \mathbb{Z}$. Therefore $2z' = z$, making z even. □

Lemma A.3. Given $x, y, z \in \mathbb{Z}$ with $z = x + y$. Let x, y be even, then z is even.

Proof. By definition: $x = 2m, \in \mathbb{Z}$ and $y = 2n, \in \mathbb{Z}$ (cf. [26]).

$$\Rightarrow\ 2m + 2n\ =\ z \tag{118}$$

$$\Leftrightarrow\ 2(m + n) =\ z \tag{119}$$

Let $(m + n) = z'$; clearly $z' \in \mathbb{Z}$. Therefore $2z' = z$, making z even. □

A.3 Execution times using plain plantri

Graph count	Vertex count	CPU time (seconds)
1	8	0.00
1	10	0.00
1	11	0.00
3	12	0.00
3	13	0.00
11	14	0.00
18	15	0.00
58	16	0.00
139	17	0.00
451	18	0.00
1326	19	0.00
4461	20	0.01
14554	21	0.04
49957	22	0.14
171159	23	0.49
598102	24	1.68
2098675	25	5.87
7437910	26	20.40
26490072	27	71.95
94944685	28	258.12
341867921	29	897.20
1236864842	30	3223.18

Table A.1: Execution times of plantri (2nd run).

Graph count	Vertex count	CPU time (seconds)
1	8	0.00
1	10	0.00
1	11	0.00
3	12	0.00
3	13	0.00
11	14	0.00
18	15	0.00
58	16	0.00
139	17	0.00
451	18	0.00
1326	19	0.00
4461	20	0.01
14554	21	0.04
49957	22	0.14
171159	23	0.49
598102	24	1.67
2098675	25	5.74
7437910	26	20.11
26490072	27	70.26
94944685	28	249.76
341867921	29	904.37
1236864842	30	3322.73

Table A.2: Execution times of plantri (3rd run).

Graph count	Vertex count	CPU time (seconds)
1	8	0.00
1	10	0.00
1	11	0.00
3	12	0.00
3	13	0.00
11	14	0.00
18	15	0.00
58	16	0.00
139	17	0.00
451	18	0.00
1326	19	0.00
4461	20	0.01
14554	21	0.04
49957	22	0.14
171159	23	0.51
598102	24	1.78
2098675	25	6.19
7437910	26	21.20
26490072	27	75.47
94944685	28	265.97
341867921	29	925.50
1236864842	30	3253.92

Table A.3: Execution times of plantri (4th run).

A.4 Execution times using plantri in sage

Graph count	Vertex count	CPU time (seconds)
1	8	0.01
1	10	0.01
1	11	0.01
3	12	0.01
3	13	0.01
11	14	0.01
18	15	0.01
58	16	0.02
139	17	0.04
451	18	0.11
1326	19	0.33
4461	20	1.15
14554	21	3.94
49957	22	14.11
171159	23	50.43
598102	24	187.21
2098675	25	701.74
7437910	26	2461.12
26490072	27	9282.16
94944685	28	33226.76
341867921	29	127207.35
1236864842	30	–

Table A.4: Execution times of plantri using sage (2nd run).

Graph count	Vertex count	CPU time (seconds)
1	8	0.01
1	10	0.01
1	11	0.01
3	12	0.01
3	13	0.01
11	14	0.01
18	15	0.01
58	16	0.02
139	17	0.04
451	18	0.13
1326	19	0.31
4461	20	1.13
14554	21	3.87
49957	22	14.11
171159	23	50.43
598102	24	182.78
2098675	25	698.52
7437910	26	2481.09
26490072	27	9252.14
94944685	28	33223.23
341867921	29	127402.41
1236864842	30	–

Table A.5: Execution times of plantri using sage (3rd run).

Graph count	Vertex count	CPU time (seconds)
1	8	0.01
1	10	0.01
1	11	0.01
3	12	0.01
3	13	0.01
11	14	0.01
18	15	0.01
58	16	0.02
139	17	0.03
451	18	0.11
1326	19	0.35
4461	20	1.13
14554	21	3.99
49957	22	14.23
171159	23	50.73
598102	24	183.51
2098675	25	706.63
7437910	26	2441.13
26490072	27	9282.56
94944685	28	33224.98
341867921	29	127201.38
1236864842	30	–

Table A.6: Execution times of plantri using sage (4th run).

A.5 Execution times of the generation of optimal 1-planar graphs

Vertex count	CPU time (s)
8	0.01
10	0.01
11	0.01
12	0.01
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.23
19	0.68
20	2.41
21	8.26
22	29.68
23	107.96
24	387.09
25	1423.47

(a) 3rd run.

Vertex count	CPU time (s)
8	0.01
10	0.01
11	0.01
12	0.01
13	0.01
14	0.01
15	0.02
16	0.03
17	0.07
18	0.23
19	0.69
20	2.44
21	8.24
22	29.25
23	102.95
24	377.58
25	1382.78

(b) 4th run.

Vertex count	CPU time (s)
8	0.04
10	0.00
11	0.01
12	0.00
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.22
19	0.68
20	2.40
21	8.03
22	29.15
23	106.61
24	382.22
25	1431.99

(c) 5th run.

Vertex count	CPU time (s)
8	0.01
10	0.00
11	0.00
12	0.00
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.22
19	0.68
20	2.38
21	8.53
22	30.86
23	112.19
24	401.36
25	1455.69

(d) 6th run.

Vertex count	CPU time (s)
8	0.01
10	0.00
11	0.00
12	0.01
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.22
19	0.70
20	2.42
21	8.25
22	29.42
23	111.43
24	395.26
25	1397.23

(e) 7th run.

Vertex count	CPU time (s)
8	0.01
10	0.00
11	0.00
12	0.01
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.22
19	0.70
20	2.54
21	8.74
22	30.56
23	108.95
24	377.13
25	1422.37

(f) 8th run.

Vertex count	CPU time (s)
8	0.01
10	0.00
11	0.00
12	0.01
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.23
19	0.67
20	2.39
21	8.07
22	28.78
23	103.31
24	400.73
25	1421.59

(g) 9th run.

Vertex count	CPU time (s)
8	0.01
10	0.00
11	0.00
12	0.01
13	0.01
14	0.01
15	0.01
16	0.03
17	0.07
18	0.22
19	0.68
20	2.42
21	8.07
22	28.90
23	105.07
24	392.24
25	1426.73

(h) 10th run.

Table A.7: Execution times of the generation of optimal 1-planar graphs.