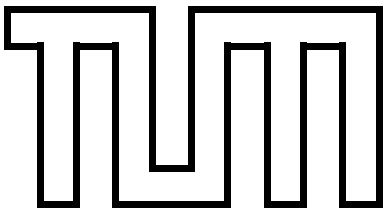
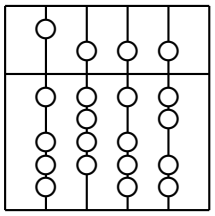




Formalizing Plane Graph Theory – Towards a Formalized Proof of the Kepler Conjecture

Gertrud Josefine Bauer

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Formalizing Plane Graph Theory – Towards a Formalized Proof of the Kepler Conjecture

Gertrud Josefine Bauer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Hans-Joachim Bungartz
Prüfer der Dissertation:	1.	Univ.-Prof. Tobias Nipkow, Ph. D.
	2.	Univ.-Prof. Dr. Jürgen Richter-Gebert

Die Dissertation wurde am 04. 07. 2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17. 01. 2006 angenommen.

Kurzfassung

Im Jahre 1998 veröffentlichte der Mathematiker Thomas Hales einen Beweis der auf Kepler zurückgehenden Vermutung, daß die kubisch dichte Kugelpackung (englisch *cubic close packing*) die dichteste Kugelpackung ist. Der Beweis beruht auf einer umfangreichen, durch ein Programm generierten Fallunterscheidung über ca. 3050 Fälle. Dabei wird jeder Fall durch einen planen Graphen mit bestimmten Eigenschaften, einem sogenannten *zahmen Graphen* repräsentiert. Wesentlich für den Beweis ist die Vollständigkeit dieser Aufzählung.

In dieser Arbeit geht es um eine Formalisierung einer Theorie planer Graphen in dem Theorembeweiser Isabelle/HOL, basierend auf einer Definition als induktive Menge: Eine Konstruktion eines Graphen beginnt mit einer einzigen Fläche, an die sukzessive neue Flächen angelegt werden. Darauf aufbauend wird ein Teil des Beweises der Kepler Vermutung formalisiert, die Vollständigkeit der Aufzählung der zahmen Graphen.

Abstract

In 1998, Thomas Hales published a proof of the Kepler Conjecture, which states that the *cubic close packing* is the densest possible packing of equally-sized spheres. The proof is by exhaustion on a set of 3050 plane graphs satisfying certain properties, called *tame plane graphs*. The enumeration of this set has been generated by a computer program, hence the completeness of this enumeration is essential for the proof.

In this thesis, we formalize a theory of plane graphs defined as an inductive set in the theorem prover Isabelle/HOL: a plane graph is constructed starting with one face and repeatedly adding new faces. Based on this theory, we formalize one part of the proof of the Kepler Conjecture, the completeness of the enumeration of tame plane graphs.

Acknowledgements

I am much obliged to Prof. Tobias Nipkow for giving me the opportunity to work on my thesis under his supervision. I thank Prof. Jürgen Richter-Gebert for acting as a referee. I am thankful to Thomas Hales who always very friendly answered my questions on his proof of the Kepler conjecture.

I owe special thanks to my current and former colleagues in Tobias Nipkows theorem proving group, Clemens Ballarin, Stefan Berghofer, Amine Chaieb, Florian Haftmann, Farhad Mehta, Steven Obua, Norbert Schirmer, Sebastian Skalberg, Martin Strecker, Tjark Weber, Markus Wenzel and Martin Wildmoser. Their helpfullness, their comments and encouragements have been extremely valuable.

Finally I like to thank my family and all friends, especially Heiko, Ulrich and Dominik for their support and patience.

Notation

We use the following Isabelle/HOL notation for lists. Given a list xs,

$[]$	is the empty list;
$x\#xs$	appends an element x in front of xs;
$[a,b,c]$	is the list containing the elements a, b, c;
$xs@ys$	is the concatenation of the lists xs and ys;
$hd\ xs$	is the head element of xs;
$last\ xs$	is the last element of xs;
$butlast\ xs$	is the list xs without the last element;
$take\ n\ xs$	returns the list of the first n elements of the list xs;
$drop\ n\ xs$	returns the list xs without the first n elements;
$rev\ xs$	is the reversed list;
$foldr\ f\ xs\ y$	repeatedly applies a function $f\ x$ on y for all $x \in set\ xs$;
$\|xs\|$	is the length of xs;
$xs[\![i]\!]$	is the element at position i in xs where $0 \leq i < \|xs\|$;
$xs[i:=a]$	is the list xs where the element at position i is replaced by a ;
$set\ xs$	is the set of elements in xs;
$[x \in xs.\ P\ x]$	is the list of all elements of xs that obey property P;
$[f\ x.\ x \in xs]$	is the list of all elements of $f\ x$ where $x \in set\ xs$;
$replicate\ n\ x$	is the list consisting of n times the element x;
$[0\ ..<\ n]$	is the list of numbers from 0 to $n - 1$;
$xs \times ys$	is a list containing all elements (x,y) where $x \in set\ xs$ and $y \in set\ ys$;
$xs \cap ys$	is a list containing all elements x with $x \in set\ xs \cap set\ ys$;
$xs - ys$	is a list containing all elements x with $x \notin set\ ys$;
$\sum x \in xs\ f\ x$	is the sum $\sum_{i=0}^{i<\|xs\|} f\ xs[\![i]\!]$;
$\cup_{x \in xs}\ f\ x$	is the concatenation of all lists $f\ x$ where $x \in set\ xs$.

We use the following Isabelle/HOL notation for arrays.

$[\![f\ x.\ x < n]\!]$	is an array consisting of elements $f\ i$ where $0 \leq i < n$;
$f[\![i]\!]$	is the element at position i in the array f.

We use the following Isabelle/HOL notation for sets. Let A and B be sets.

$\{\}$	is the empty set;
$a \in A$	means *a is contained in A*;
$\{a,\ b,\ c\}$	is the set of elements a, b, c;
$A \cup B$	is the union of A and B;
$A \cap B$	is the intersection of A and B;
$A - B$	is the difference of A and B;
$A \times B$	is the Cartesian product of A and B;
$A \subseteq B$	is the subset relation of A and B;
$A \subset B$	is the proper subset relation of A and B;
$A//R$	is the quotient of a set A by an equivalence relation R;
$\{x.\ \ P\ x\}$	is the set of all elements x with $P\ x$;
$\{f\ x \mid x.\ x \in A\}$	is the set of all elements $f\ x$ with $x \in A$;
$\forall\, x \in A.\ P\ x$	means *for all elements x in A holds P x*;
$\exists\, x \in A.\ P\ x$	means *there exists an element x in A with P x*.

We use the following mathematical notation.

$\mathbf{N}$	is the set of natural numbers;
$\mathbf{R}$	is the set of real numbers;
$\mathbf{R}_0^+$	is the set of non-negative real numbers;
$\mathcal{V}$	is the set of vertices in a graph;
$\mathcal{E}$	is the set of edges in a graph;
$\mathcal{F}$	is the set of faces in a graph;
$A^{(2)}$	is the set two element-sets (unordered pairs) of A.
$f_1 \cong f_2$	is the equivalence relation of two faces f_1 and f_2;
$g_1 \simeq g_2$	is proper isomorphism of two graphs g_1 and g_2;
$g_1 \cong g_2$	is isomorphism of two graphs g_1 and g_2;

We use the following notation to indicate if lemmas or theorems have been formally proved in Isabelle/HOL or not yet.

$!$	is the symbol for lemmas not yet formally proved in Isabelle;
$\surd$	is the symbol for proved lemmas in Isabelle.

Contents

Chapter 1

Introduction

The traditional understanding of a rigorous mathematical proof is a sequence of steps derived from an accepted sets of axioms, such that a reader can comprehend each step, or even such that each step can be reduced to basic logical rules. It is an argument suitable to convince a reader of the correctness of a statement.

In the last decades some proofs of mathematical theorems were published that do not fulfill these strict criteria, because they heavily rely on complex computer computations. One example is the proof of the Four-Color-Conjecture by Kenneth Appel and Wolfgang Haken 1977 [1], [2], [3], improved by Neil Robertson, Daniel Sanders, Paul Seymour and Robin Thomas 1996 [30], [32]. Another example is the proof of the Kepler Conjecture, presented by Thomas C. Hales 1998 [17]. Both proofs are by exhaustion and the large set of cases to be analyzed was generated and checked by a computer program. Hence one part of the proof is to verify every case, the other part is to prove that the cases are exhaustive, i.e. to show the completeness of the computer program generating the cases. Of course it is infeasible for a human to verify the correctness of a computer program by hand, due to the complexity of possibilities to be considered. This is the reason why to some mathematicians such a proof is not quite acceptable, as the proof cannot be reviewed. This motivates computer-aided verification of these programs.

In 1998, Thomas C. Hales of the University of Michigan announced that he found a proof of the Kepler conjecture, making extensive use of computer calculations. At that stage it consisted of 250 pages of notes and 3 gigabytes of computer programs, data and results [17] (including [12], [18], [15], [16],

[19], [20], [11], [21]). Later, Hales published an overview of the proof in [23], which has been updated [24] and submitted for the Annals of Mathematics. A committee of 12 referees was constituted in order to check correctness of the proof. After four years, Gabor Fejes Tóth, the head of the referees, reported that they were "99% certain" of the correctness of the proof, but they could not certify the correctness of all of the computer calculations. Anyhow, the Annals now have enough confidence in the proof and will publish it in [25].

The Kepler Conjecture states that the way oranges are usually stacked at the market (in a so called *cubic close* sphere packing) is the densest possible way. A packing is represented by a finite set of sphere centers. The first part of

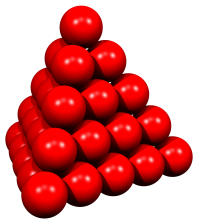


Figure 1.1: Cubic close packing

the proof presented by Hales is to reduce this infinite problem to a finite one, i.e. to show that it suffices to fix one sphere and consider only the spheres which are 'close' to this sphere. Then, the set of close spheres is represented by a plane graph, i.e. a graph embedded in the plane, characterizing which of those again are close to each other. All graphs corresponding to possible counterexamples of the Kepler Conjecture have certain properties, summarized in a property *tame*. This set of tame plane graphs is finite and can be enumerated by a computer program. The completeness of this enumeration is essential for the proof of the Kepler conjecture. It reduces the proof to the analysis of all tame plane graphs: it has to be verified that every tame plane graph gives rise to a packing with density not greater than the density of the cubic close packing.

In response to the difficulties in verifying his proof, in January of 2003, Hales initiated the project "**F**lys**p**ec**k**" ("**F**ormal **P**roof of **K**epler") in an attempt to use computers to verify every step of the proof [14]. In English, *flyspeck* can also mean *examine closely*. Hales expects the project to take about 20 person-years of work. This project has also attracted attention in popular print media (Science [27],[26] and Nature [31]).

Similar efforts have been made in the verification of the Four-Color-Conjecture by Georges Gonthier [13], who finally completely proved the Four-Color-Conjecture in the theorem prover Coq [9]. A case-study on the Five-Color-Theorem has been carried out in Isabelle/HOL, which is based on the same mathematical background as the Four-Color-Conjecture, but it does not need this extensive case analysis used in the proof of the Four-Color-Conjecture.

The aim of this work is a formalization of one part of the proof: the completeness of the enumeration of the set of tame plane graphs, using the theorem prover Isabelle/HOL. This is based on a formalization of plane graphs that can also be used for further applications on the theory of plane graphs.

1.1 Contributions

Our starting point is a Java program written by Thomas C. Hales, which computes a certain class of plane graphs where each one represents a case in Hales' proof.

- We translate Hales' Java program for enumerating all tame plane graphs to an Isabelle/HOL function *Enumeration*. We introduce a data structure for graphs, which is similar to the pointer structures in Hales' Java program, but uses lists of values instead of references. We define functions by primitive recursion in the higher-order logic of Isabelle/HOL. We define plane graphs by an inductive set and formalize the notion of tame graphs as a HOL predicate.

- We show how completeness of *Enumeration* can be proved by proposing a proof structure in Isabelle and partially carry out the proof in Isabelle. The completeness proof yields the result that all tame plane graphs are generated by *Enumeration*.

- As next step, we prove completeness of Hales' Java program, i.e. that the Java program completely enumerates all tame plane graphs. Using Isabelle's code generation facility [6], which is one of the trusted parts of Isabelle, we generate executable ML code, execute it and compare the output with the output of Hales' Java program. Hence we obtain a confirmation that all plane graphs are generated by Hales' Java program.

- Finally we validate our formalization: we show that our definition of plane graphs really represents all plane graphs.

1.2 Overview

In Chapter 2, we give a short overview of the proof for the Kepler Conjecture and introduce some mathematical background for the graph theoretic part of the Kepler Conjecture. Moreover, we discuss possible formalizations of plane graphs in theorem provers and introduce Isabelle/HOL notation used for definitions and proofs.

We have implement one of these formalizations in the theorem prover Isabelle/HOL. This is presented in Chapter 3. We introduce data structures and construction functions for faces and graphs and prove their correctness. We give the abstract definition of plane graphs and their implementation as inductive set and derive an induction principle. We define isomorphims on plane graphs and implement an executable isomorphism test.

In Chapter 4, we recall the definition of tameness from Hales' proof of the Kepler Conjecture and present its formalization in Isabelle/HOL.

In Chapter 5, we introduce a sequence of refinement steps starting from the definition of plane graphs, succeedingly restricting the set of graphs by imposing the restrictions of tameness, ending up with a finite set of graphs *Enumeration*.

We present a proof structure of the completeness theorem induced by a set of refinement steps of the enumeration algorithm. Chapter 6, contains the completeness proofs.

Finally, in Chapter 7 we summarize the results and discuss future works. We present some basic algorithms in Appendix A and some readable Isabelle/Isar proof texts in Appendices B and C.

Chapter 2

Preliminaries

In Section 2.1, we give an overview of the proof of the Kepler conjecture.

In Section 2.2, we introduce planar and plane graphs and briefly present properties of plane graphs that have to be reflected in a formalization. In Section 2.3, we describe a formalization of plane graphs in terms of sets of faces. In the following Section 2.4, we discuss alternative formalizations of plane graphs, which are also suitable for verification in a theorem prover: using triangulations (see Section 2.4.1) or oriented combinatorial maps (see Section 2.4.2). Next, we compare some aspects of these formalizations with regard to adequacy for formalization in a theorem prover.

Finally, in Section 2.5 we introduce Isabelle/HOL notation used for definitions and proofs.

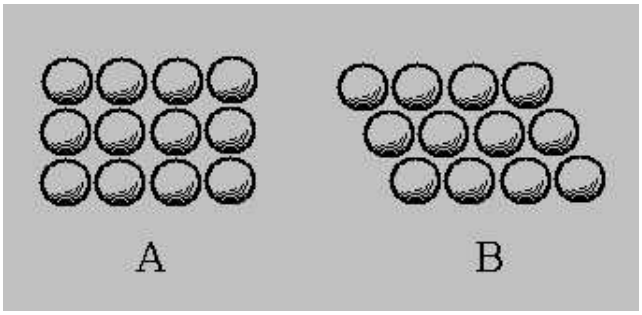
2.1 The Kepler Conjecture

In 1611, Kepler proposed that the *cubic close packing* (see Figure 1.1) is the densest possible packing of equally-sized spheres.

Theorem (Kepler Conjecture)
The cubic close packing of unit spheres has optimal density.

2.1.1 History

In 1611, the german astronomer Johannes Kepler (1571-1630) describes packings of equally sized spheres in the space in a small booklet *strena seu de nive sexangula* ("About the hexagonal snowflake"). Spheres can be arranged in a plane in two different ways: either in a *square* (A) or in a *triangular* arrangement (B). In the first case every sphere is touched by 4 surrounding spheres, in the second by 6 spheres.



An arrangement of spheres in the space is constructed by putting layers of the kind A on top of each other, such that in the higher layers every sphere lies on four spheres of the layer below. Hence, every sphere is touched by 4 spheres of the same layer, 4 spheres of the layer below and 4 spheres of the layer above. This packing is now known as the *cubic close packing* (ccp). This is the way oranges are typically stacked at the market.

Kepler made the following assertion, which is now known as the Kepler Conjecture:

> *Coaptatio fiet arctissima, ut nullo praeterea ordine plures globuli in idem vas compingi queant.*

In English:

> *This packing will be the densest, such that in no other arrangement could more spheres be stuffed into the same container.*

2.1.2 Cubic Close Packings

Putting layers of kind B on top of each other, forming a triangular pyramid (with triangular base), leads to a second kind of packing.

Both packings are identical up to rotation: if we consider a square pyramid (with square base) formed by layers of spheres of kind A and if we consider one side of this pyramid, we observe that it is formed by layers of spheres of type B. If we take this layer as the base layer of the packing and compare the two packings, we observe that the packings are in fact identical.

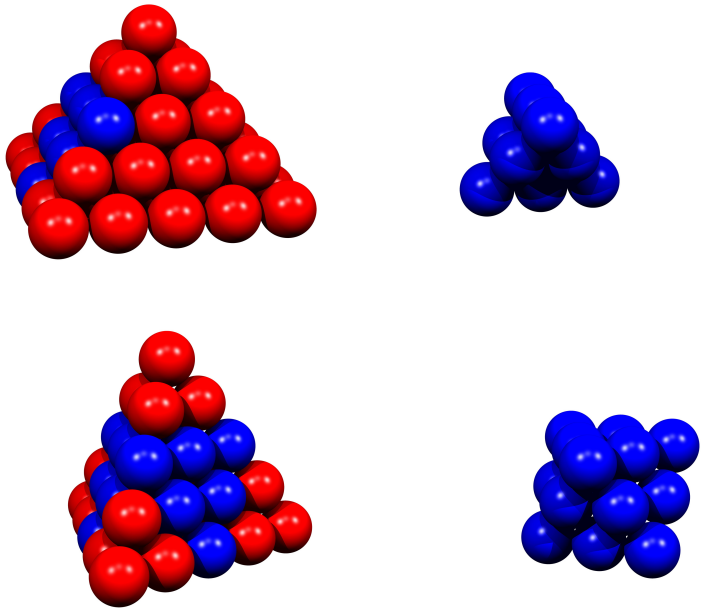


Figure 2.1: Face centered cubic packing

There are always 2 possible ways to put a layer of kind B on top of another. There are 3 different positions of the layers (see Figure2.2), called 1, 2, 3.

Any sequence of layers, where any two subsequent layers differ (like for example 1, 2, 1, 3, 2, ...) leads to a packing of the same density, since shifting the layers does not change the density of a packing. Along all these packings, the cubic close packing is the one with a cyclic sequence of layers (1, 2, 3, 1, 2, 3, ...). An other one is the *hexagonal close packing* (hcp), which has a sequence of layers (1, 2, 1, 2, 1, 2, ...).

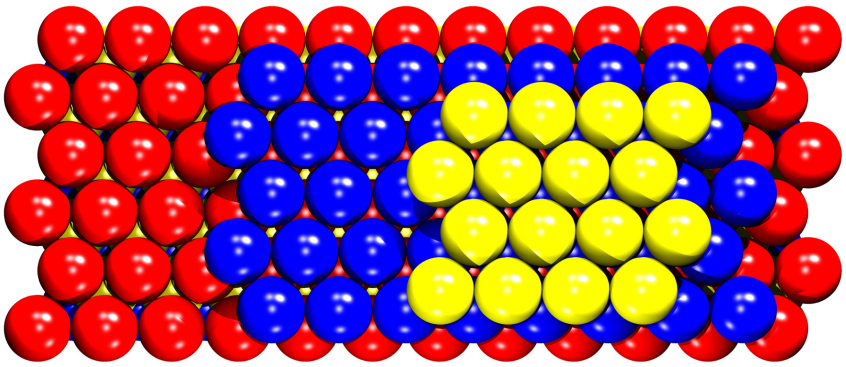


Figure 2.2: 3 different layers in the face centered cubic packing

After Kepler's assertion, many centuries passed without a rigorous proof. When Hilbert proposed a list of 23 open problems to the mathematics community in 1900, he included the Kepler Problem ("What is the densest packing of equal sized spheres in space?") as part of the 18th problem.

2.1.3 Density

A *packing* is an arrangement of congruent spheres that are non-overlapping in the sense that interiors of the spheres are pairwise disjoint. We may assume that the radius of the spheres is 1, since the density is independent of the radius. We may also assume that the packing is *saturated*, i.e. that there is no space to add further spheres, because otherwise the density of the packing is not maximal. A saturated packing is represented by the set of centers Λ of the spheres in the packing. The choice of radius of 1 implies that any two points in Λ have a distance of at least 2 from each other.

The *density* of a packing is the ratio of the space occupied by the spheres to the total space. We first define *finite density* $\delta(x, r, \Lambda)$ as the ratio of the space occupied by the spheres of the packing within a sphere of radius r at center x.

$$\delta(x, r, \Lambda) = \frac{vol(P(\Lambda) \cap S(x, r))}{vol(S(x, r))}$$

where $P(\Lambda)$ is the union of all spheres in the packing Λ, and $S(x, r)$ is a sphere at center x with radius r.

The *density* $\delta(\Lambda)$ of a packing Λ is then defined as the limit of *finite densities* with increasing radius r.

$$\delta(\Lambda) = \lim_{r\to\infty} \delta(x, r, \Lambda) \text{ for an arbitrary point } x.$$

A *Voronoi cell* around a point $x \in \Lambda$ is the set of all points in the space that are closer to v than to any other center point of Λ. The volume of each Voronoi cell in the cubic close packing is $\sqrt{32}$, hence the density is $\frac{\sqrt{32}}{4/3\pi} = \frac{\pi}{3\sqrt{2}} \approx 74.048\%$.

2.1.4 Hales' Computer Based Proof

Structure of Hales' proof of the Kepler Conjecture (by classical contradiction)

1. The first part in the proof is the reduction of an infinite problem (infinitely many variables representing infinitely many positions of spheres in the space) to a finite one. To this aim, a so called *decomposition star* is constructed around the center v of one sphere in a packing, depending only on the set Λ of all sphere centers in the packing. Hales defines a continuous function σ on the space of all decomposition stars in [12]. Hales shows that the Kepler conjecture can be reduced to the statement

   $$\sigma(D) \leq 8 \quad \text{for all decomposition stars } D.$$

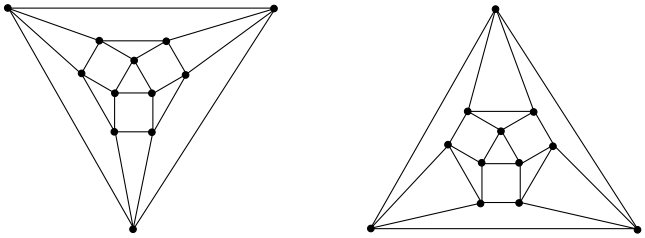
2. To every possible counterexample D ('*contravening decomposition star*'), a *plane* graph ('*contravening* plane graph') (see Chapter 3) is associated in the following way:

   We fix one sphere s in the packing and consider all other spheres of a packing that are *close* to s. Two spheres in a packing have minimal distance 2. Two spheres are called *close* if the distance of their centers is not greater than a fixed constant 2.51. For every sphere close to s, we draw a vertex on the surface of s, projecting its center in direction of the center of of s.

   Two vertices are connected by an edge if the corresponding spheres are close to each other. In this way a planar graph is drawn on the surface

of s. Intersecting lines are avoided by the maximal distance of 2.51, which is proved in [22]. This graph can also be drawn in the plane, e. g. using stereographic projection.

Example
The following plane graphs are associated with the cubic close packing and the hexagonal close packing:



3. All counterexamples have certain properties. These properties are reflected in properties of the contravening plane graphs and collected in a notion of *tame plane graphs* (see Chapter 4).

   Theorem
   Every contravening plane graph is tame.

   This theorem reduces the search for a contravening plane graph to the set of tame plane graphs.

4. A superset of the set of tame plane graphs is enumerated by a computer program. From the output of the program, a list of plane graphs is created, called the *archive*, which contains about 3050 graphs.

   Theorem (Completeness of Enumeration)
   Every tame plane graph is isomorphic to a graph in the archive.

   This reduces the proof of the Kepler conjecture to the analysis of the decomposition stars attached to the finite list of graphs in the archive.

5. For every decomposition star it must be verified that it is not contravening, using linear programing solving equations with 100 to 200 variables and 1,000 to 2,000 constraints. This leads to the conclusion

   Theorem
   Every graph in the archive is not contravening.

Qed

2.2 Planar Graphs

The usual definition of planar graphs uses an embedding in the plane, represented by closed non-intersecting curves, which are continuous function from an interval $[0, 1]$ to $\mathbf{R}^2$. A difficulty of this definition is that a lot of mathematical background (e. g. on continuous functions) is required. Moreover, a particular graph can be drawn in many different ways in the plane. Hence an abstraction identifying these equivalent embeddings is beneficial for a verification.

Definition
A *graph* $g = (\mathcal{V}, \mathcal{E})$ is a finite, nonempty set of *vertices* $\mathcal{V}$ and a set of (unoriented) *edges* $\mathcal{E}$, such that $\mathcal{E}$ is a subset of $\mathcal{V}^{(2)}$, the set of two-element sets (unordered pairs) of $\mathcal{V}$ [8].

By definition, graphs do not contain *loops* (i. e. edges joining a vertex to itself) or *multiple edges*, (i. e. several edges joining the same two vertices).

loops:  multiple edges: 

Definition
We say that an edge $\{v, w\}$ *joins* the vertices v and w. Two vertices v and w are called *adjacent* or *neighboring* in g if $\{v, w\} \in \mathcal{E}$. Both vertices v and w are called *incident* with $\{v, w\}$.

A graph $g' = (\mathcal{V}', \mathcal{E}')$ is a *subgraph* of $g = (\mathcal{V}, \mathcal{E})$ iff $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$.

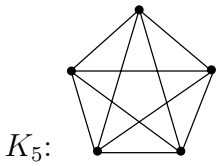
A graph with n vertices and $\binom{n}{2}$ edges is called a *complete* n-graph and is denoted by K_n.

The *degree* of a vertex is the number of adjacent vertices.

A *path* of length l is a graph of the form $p = (\mathcal{V}_p, \mathcal{E}_p)$, where $\mathcal{V}_p = \{v_0, v_1, \ldots v_l\}$ and $\mathcal{E}_p = \{\{v_0, v_1\}, \{v_1, v_2\} \ldots \{v_{l-1}, v_l\}\}$. v_0 and v_l are the *endpoints* of p and we say that p is a path *from* v_0 *to* v_l. A path of distinct vertices of length n whose endpoints coincide is called an *n-cycle*.

Definition
A *planar* graph is one that can be *embedded* in the plane i. e. it can be drawn in the plane such that no two edges intersect. An *embedding* of a graph $g = (\mathcal{V}, \mathcal{E})$ is a function from $\mathcal{V}$ to $\mathbf{R}^2$ such that for every edge $\{v, w\} \in \mathcal{E}$ we can connect the points associated with v and w by non-intersecting curves.

Example
The complete graph K_5 of five vertices is not planar.



Note that an embedding in the plane is not unique. This is illustrated by the following example.

Example
It is always possible to arrange the points of a planar graph such that the vertices are connected by straight lines (see Figure 2.3 (a)). Given an embedding, any face can be chosen to be the exterior one (see Figure 2.3 (b)). The set of faces is not necessarily unique (see Figure 2.3 (c)). Given an embedding, the mirrored embedding is also an embedding for the same graph (see Figure 2.3 (d)).

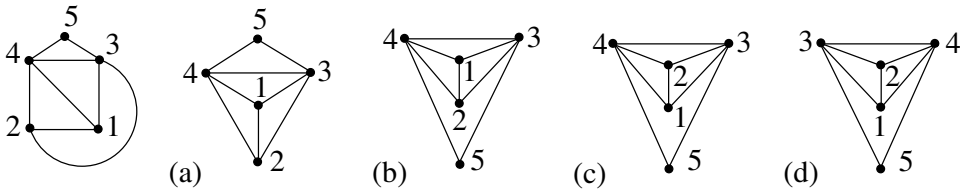


Figure 2.3: Different embeddings of a graph in the two dimensional plane

Definition
A *plane* graph is one that *is* embedded in the plane, i. e. a graph together with an *embedding*. If we remove all points from the plane which are associated to the vertices and edges of a plane graph, the remainder is divided into connected components, called *faces*. Every face can be represented by the set of its bounding edges, which induce a cyclic subgraph of g. Then an embedding can be represented by a set of faces. Hence a plane graph g can be represented by $g = (\mathcal{V}, \mathcal{E}, \mathcal{F})$, where $\mathcal{V}$ is the set of vertices, $\mathcal{E}$ is the set of edges, and $\mathcal{F}$ is the set of faces.

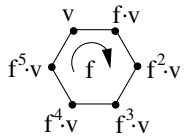
2.3 A Formalization of Plane Graphs

In this section, we give an abstract mathematical definition of graphs in terms of sets of faces. We follow the graph representation used in Hales' Java program. In Section 3.1, we present the implementation of these concepts in Isabelle/HOL.

Definition
A *face* f is a finite set of vertices V_f of cardinality at least 3, together with a cyclic permutation on then. We write this permutation as $v \mapsto f{\cdot}v$. Consequently, $f{\cdot}v \neq v$ for all $v \in V_f$.

In order to draw graphs, we need to decide on a convention on the orientation in which faces are to be drawn. We draw faces in clockwise orientation (see the following example).

Example
A face of length 6.



Definition
An *unoriented edge* is a two-element set $\{v, w\}$ of vertices such that $f{\cdot}v = w$ for some face f. An *(oriented) edge* is a pair (v, w) such that $f{\cdot}v = w$ for some face f. We usually mean oriented edges when we refer to edges.

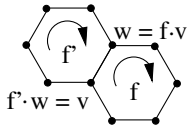
Definition
A *plane graph* g is a nonempty finite set of *faces* with the following properties:

1. For a face f in a graph g, if $f{\cdot}v = w$ then there is a unique face f' in g, with $f'{\cdot}w = v$.

   Hence an automorphism of the faces of g incident with v is associated to each vertex v in g. We write $f \mapsto (g, v){\cdot}f = f'$ for this function.

   Note that the convention of drawing the vertices in a face in clockwise orientation implies that the automorphim $f \mapsto (g, v){\cdot}f$ permutes the faces around a vertex v in counterclockwise orientation.

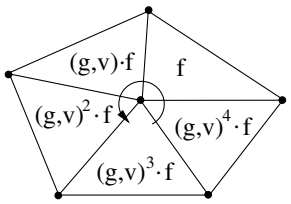
Moreover, each unoriented edge occurs in exactly two faces of g with opposite orientation. That is, for a face f and an edge in e there is exactly one face f' that contains the edge with opposite orientation.



2. For each vertex, the function $f \mapsto (g, v) \cdot f$ is a cyclic permutation of the set of faces containing v.

3. Euler's formula holds, relating the number of vertices $|\mathcal{V}|$, the number of edges $|\mathcal{E}|$ and the number of faces $|\mathcal{F}|$

$$|\mathcal{V}| - |\mathcal{E}| + |\mathcal{F}| = 2$$



Definition
The *length* $|f|$ is the number of vertices in a face f. A face of length 3 is called a *triangle*, a face of length 4 is called a *quadrilateral*, faces of length at least 5 are called *exceptional*. The *degree* of a vertex is the number of faces containing the vertex itself. $tri(v)$ is the number of triangles containing a vertex v. $quad(v)$ is the number of quadrilaterals containing a vertex v. $except(v)$ is the number of exceptionals containing a vertex v. The *type* of a vertex is a triple (p, q, r), where p is the number of triangles, q the number of quadrilaterals, and r is the number of exceptional faces containing the vertex. We write $type(v) = (p, q)$ as a shorthand for $type(v) = (p, q, 0)$.

Definition
Two graphs g and h are called *properly isomorphic* ($g \simeq h$) if there is a bijection of vertices, inducing a bijection of faces. For each graph g there is an *opposite graph* g^{op} obtained by reversing the cyclic order in each face. A graph g_1 is called *isomorphic* to another graph g_2, if g_1 is isomorphic to g_2 or g_2^{op}.

For the construction of plane graphs we distinguish two different kinds of faces: A face is marked either *final* or *nonfinal* . We use the terminology of the Java program [23], whereas in Hales' overview paper [24] faces are called *complete/incomplete*. . In figures we draw final faces *white* and nonfinal faces *grey* (except a nonfinal face at the outside of a graph). Nonfinal faces are temporary faces during a construction of a graph. They can be further refined by adding new faces, whereas final faces may not be changed any more. A graph is called *final* if all faces are final, otherwise it is called *nonfinal* or *partial*.

2.4 Alternative Formalizations

In the following section we compare two other approaches to formalize the theory of plane graphs.

2.4.1 Formalization by Triangulations

This section describes a way to formalize plane graphs, as subgraphs of triangulations. We first define triangulations inductively and obtain plane graphs by omitting some edges. This approach was proposed by John Harrison. A case study on the proof of the five-color-theorem based on this formalization has been carried out in Isabelle/HOL [4].

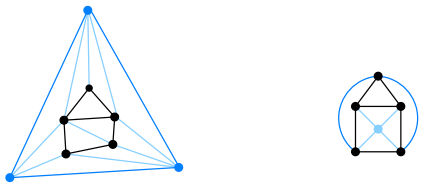


Figure 2.4: Example: A planar graph embedded in triangulations

A graph is *planar* iff it is a subgraph of a *triangulation*, i.e. if it can be extended to a triangulation by only adding edges (see Figure 2.4). Note that this triangulation is not unique: a graph may be embedded in different triangulations.

Representation The representation of graphs is as follows:

Vertices are elementary objects. Edges are sets of 2 vertices. Faces are sets of vertices. A (plane) graph is represented by a set of faces. We distinguish between *interior* faces and the *boundary* (the *exterior* face) of a graph. Interior faces contain exactly 3 vertices, whereas the exterior face may contain an arbitrary number of vertices.

Definition
Triangulations are plane graphs where every face (including the exterior face) is a triangle. Triangulations are defined in terms of *near triangulations*. Near triangulations are plane graphs where every face except the exterior face is a triangle. Near triangulation can be defined as an inductive set: We start the

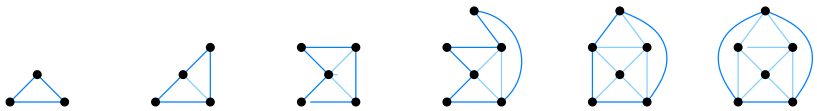


Figure 2.5: Example: Construction of a triangulation

construction with one triangle. A new triangle face can be added to the set of faces of a graph by attaching it to the exterior face of a graph (see Figure 2.6). We can attach triangles $\{a, b, c\}$ in two different ways: if there is an edge $\{a, c\}$ on the boundary, we can add a new vertex b and edges $\{a, b\}$ and $\{b, c\}$ (Type I). If there are two edges $\{a, b\}$ and $\{b, c\}$ on the boundary, we can add an edge $\{c, a\}$ (Type II). Explicitely excluded are triangles where two new edges $\{a, b\}$ and $\{b, c\}$ must be added and the vertex b is one of the vertices already constructed (Type III), because otherwise the boundary may become a set of cycles during the construction.

A triangulation can finally be defined as a near triangulation with a triangle exterior face.

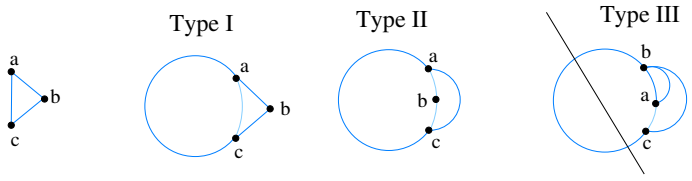


Figure 2.6: Inductive definition of near triangulations

The advantage of this definition is that we get an induction principle for free

which allows proofs on the construction of a triangulation. For example we can prove by induction that every edge is contained in exactly two faces. Another example of a property proved by induction for all generated graphs is *Euler's formula*.

There are many different ways to construct a given triangulation. Moreover, we can start the construction with any of its faces. This property can be stated as follows:

Lemma
If we can construct a certain (near) triangulation by successively adding the faces $f_0, f_1, f_2, \ldots, f_m$, then we can generate the same (near-)triangulation by another valid sequence of faces starting with f_i, where $i \in \{0 \ldots m\}$.

Note that not every permutation of faces is allowed for the construction: at any time in the construction the graph must be connected and particularly the boundary must be a single simple cycle. A generalization which implies the above lemma is the following property:

Lemma
If we can both construct near triangulations g and h and h is a subgraph of g, then we can reach g by some construction starting with h.

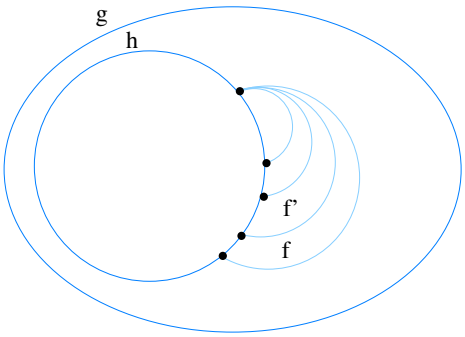
The restriction to only one boundary complicates the following proof, since it restricts the order in which faces may be added.

Proof
We prove the lemma by induction on the cardinality of the set difference $g - h$.

Base case: if $h = g$ the assertion is trivially true.

Induction case: $h \subset g$. We assume that the hypothesis holds for all subgraphs h' of g with $|g - h'| < |g - h|$ and particularly for all graphs h', where $h \subset h'$. Thus, is suffices to show that for any subgraph h we can find a face $f \in g - h$ that can be added.

It is always possible to find a face f that lies in $g - h$ and contains an edge of the boundary of h: every edge in a near triangulation is contained in exactly two interior faces, except boundary edges which are contained in exactly one interior face. Since $g \neq h$, there must be a boundary edge of h that is not a boundary edge of g. This edge is contained in two interior faces of g but only one face of h.

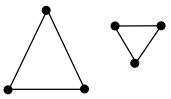


If f is of Type I or II for the graph h then it can be added, but not if f is of Type III. Hence we need to find a different face that can be added. We will find one in the region of g that is enclosed by f and h: first, we need to introduce the notion of the interior of a cycle, the set of faces enclosed by a circle. If f is of Type III, then there must be a face f' enclosed by a cycle consisting of edges of f and the boundary h, and sharing an edge with the boundary of h. Take the face f'' with the minimal set of faces enclosed by f'' and h. This set cannot contain any faces of Type III (contradiction to minimality), but it must contain at least one face that shares an edge with h. This face can be added to h. We obtain a graph h' with $h \subset h'$, hence we can reach g from h'. **Qed**

2.4.2 Formalization by Combinatorial Maps

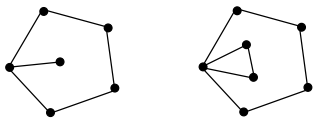
Another way to represent planar graphs is by *planar (oriented) combinatorial maps* [33, 10]. This formalization was used in Gonthier's proof of the four-color-theorem [13].

Representation We restrict ourselves to connected graphs. We exclude disconnected graphs and especially graphs with isolated vertices, i. e. vertices that are joined with no other vertex (vertices of degree 0):

Disconnected graphs:  Isolated vertex: •

Moreover, we assume that no vertex appears twice in a face. Every face is isomorphic to a polygon and the length of every face is at least 3. Hence, for

example, vertices of degree 1 are excluded:



The basic elements of (oriented) combinatorial maps are *half-edges* (*directed edges*). Consider a set of half-edges T.

Definition
An *(oriented) combinatorial map* is a pair (α, σ) of permutations of T such that

1. the permutation group generated by the permutations $\{\alpha, \sigma\}$ is transitive.
2. $\alpha^2 = I$, and
3. $\alpha(e) \neq e$ for all $e \in T$.

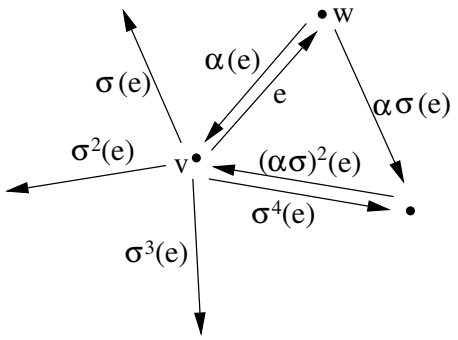


Figure 2.7: Oriented combinatorial maps

The cycles of α are called *edges*, the cycles of σ are called *vertices*. The permutation α associates to every edge the inverse edge. The permutation σ permutes the half-edges around of a vertex. A vertex v is *incident* with an edge e iff e and v share a half-edge. Two vertices v and w are *adjacent* iff both v and w are incident with a common edge.

The composition $s_1 s_2$ of two permutations s_1, s_2 is defined by $(s_1 s_2)\ e = s_2(s_1(e))$ ("first s_1, then s_2"). The cycles of the composition $\alpha\sigma$ of α and σ

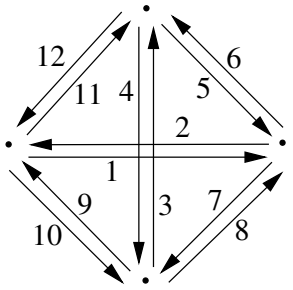
are called *faces*. A graph *graph*(m) is associated with every oriented map $m = (\alpha, \sigma)$: vertices in maps correspond to vertices in graphs, edges in maps correspond to edges in graphs, and faces in maps correspond to faces in plane graphs.

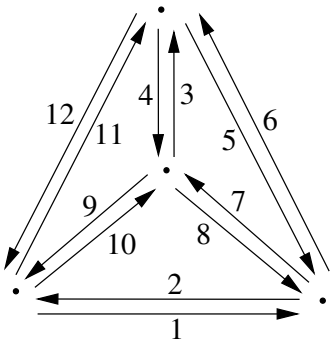
Definition
The *Euler characteristic* is the number $|\mathcal{V}|-|\mathcal{E}|+|\mathcal{F}|$, where $|\mathcal{V}|$ is the number of vertices, $|\mathcal{E}|$ is the number of edges, and $|\mathcal{F}|$ is the number of faces.

Definition
An oriented map is *plane* if its Euler characteristic is 2. Hence a plane map is an abstraction for a (non-intersecting) embedding of a graph in the plane. A plane graph is associated with every plane oriented map. A graph is planar if it is isomorphic to some graph of a plane map.

Example
Two oriented maps for K_4, the complete graph with four vertices.
The first map is not plane:
$\alpha = (1,2)(3,4)(5,6)(7,8)(9,10)(11,12)$, hence $|\mathcal{V}| = 6$
$\sigma = (12,4,5)(6,2,7)(10,1,11)(8,3,9)$, hence $|\mathcal{E}| = 4$
$\alpha\sigma = (1,7,3,5,2,11,4,9)(12,10,8,6)$, hence $|\mathcal{F}| = 2$

The second map is plane:
$\alpha = (1,2)(3,4)(5,6)(7,8)(9,10)(11,12)$, hence $|\mathcal{V}| = 6$
$\sigma = (12,4,5)(6,7,2)(1,10,11)(8,3,9)$, hence $|\mathcal{E}| = 4$
$\alpha\sigma = (1,6,12)(2,10,8)(3,5,7)(4,9,11))$, hence $|\mathcal{F}| = 4$





2.4.3 Summary

In this section we compare different aspects of formalizations. We discuss advantages and disadvantages of their application in theorem provers.

Inductive Definitions have the advantage that we obtain an induction principle on the construction, which we can use to prove properties about graphs. A problem of inductive definitions is that the construction implicitly defines an order in which the faces are inserted. For reasoning about graphs we need to abstract from this order and be able to reorder the faces such that we obtain an isomorphic graph.

Interior Faces/Single Exterior Faced vs. Final Faces/Nonfinal Faces: The approach with one outer face has the disadvantage that faces of Type III may not be allowed. This restricts the possibilities of adding new faces. The approach with final/nonfinal faces allows more reorderings of faces, which allows for simpler proofs. However, arbitrary reorderings of faces are not possible, either: at any step in the construction the graph must be connected.

Triangles vs. Polygons: The definition using triangles is by far easier. This simplifies proofs about the construction. If faces of arbitrary length can be added, the graph modifications and proofs about those are more complex. On the other hand, a formalization based on triangles requires operations to delete edges in order to construct non-triangular graphs, which complicates a verification.

Directed vs. Undirected Edges: Directed edges make it possible to distinguish the interior and exterior of a cycle. This is required to formalize a discrete version of the *Jordan Curve Theorem*.

Redundant Data Structures can improve the efficiency of algorithms. For example, we can extend the representation of plane graphs $g = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ by a mapping $\mathcal{V} \to \mathcal{P}(\mathcal{F})$ that stores for every vertex $v \in \mathcal{V}$ the set of faces that contain v. The disadvantage is that we need consistency lemmas about the representation, which can be proved by induction on the construction.

The following table summarize these aspects for the different formalizations. It shows that the formalizations based on directed faces and combinatorial

maps are quite similar, whereas the formalization based on triangulation differs significantly:

	Directed Faces	Triangulations	Combinatorial Maps
Inductive Definition	Interior / Exterior Faces	Final / Nonfinal Faces	No Inductive Definition
Faces	Polygons	Triangles	Polygons
Edges	directed	undirected	directed

2.5 Isabelle

The formalization of plane graphs and tameness was carried out in Isabelle/HOL. We give a short overview of the basic concepts of the object logic HOL, which are used in this thesis. For an introduction to Isabelle/HOL see [29].

Basic Types and Functions

Proof rules are expressed in the meta-logic of Isabelle/HOL, consisting of meta implication $\Longrightarrow$ and meta-quantification $\bigwedge x.\ P\ x$. We write $A \Longrightarrow B \Longrightarrow C$ for $A \Longrightarrow (B \Longrightarrow C)$. Functions may be defined using definitional equality $\equiv$, or as recursive functions, for example on an inductive datatype. We use the following standard functions of the Isabelle/HOL Library.

Bool The type *bool* of all logical formulas contains the elements *True* and *False*. The following connectives are defined: $\wedge$ (*and*), $\vee$ (*or*), $\neg$ (*negation*) and the (overloaded) $=$ on booleans (‘*if and only if*)

Moreover, $\forall x.\ P\ x$ means *for all elements* x *holds* $P\ x$; $\exists x.\ P\ x$ means *there exists an element* x *with* $P\ x$; and $\exists !x.\ P\ x$ means *there exists a unique element* x *with* $P\ x$.

Sets The type *'a set* is the type of all sets of elements of type *'a*. Here *'a* is a type variable. We write $a \in A$ if an element a is contained in the set A. The empty set is denoted by $\{\}$. A particular set may be denoted like $\{a, b, c\}$. The union of two sets A and B is denoted by $A \cup B$; set intersection is denoted by $A \cap B$; set difference is denoted by $A - B$; the Cartesian product

is denoted by $A \times B$. The subset relation is denoted by $A \subseteq B$; a proper subset relation is denoted by $A \subset B$.

$\forall\, x \in A.\ P\ x$ means *for all elements x in A holds P x*; $\exists\, x \in A.\ P\ x$ means *there exists an element x in A with P x*. Set comprehension $\{a.\ P\ a\}$ is used to construct the set of all elements with property P; $\{f\ x \mid x.\ x \in A\}$ is the set of all elements $f\ x$ with $x \in A$ The notation $A//R$ denotes a quotient construction of set A by an equivalence relation $R \subseteq A \times A$.

Functions The type $'a \Rightarrow 'b$, is the type of all functions of elements of type $'a$ to elements of type $'b$. Function application is written in curried style: $f\ x$ is the function f applied to an argument x. We use λ-notation $\lambda x.\ f\ x$ for a function which assigns to an argument x the value $f\ x$.

Natural Numbers and Integers The type *nat* of natural numbers is an inductive datatype with the constructors *0* and *Suc*. We denote particular natural numbers like *0*, *1*, *2*, *3*, *4*, ..., the type *int* is the type of all integers. We denote particular integers like ..., -2, -1, *0*, *1*, *2*, ... On both types we use $+$ (addition), $-$ (subtraction), $*$ (multiplication), $<$, $\leq$, $=$ (comparisons), and *mod* (modulus). Primitive recursive functions on natural numbers are defined in the following style.

$f :: nat \Rightarrow 'b$
$f\ 0 = ...$
$f\ (Suc\ n) = ...\ (f\ n)\ ...$

Lists We widely use lists in our formalization in order to obtain an executable algorithm out of the definition. On the one hand we use lists to represent a permutation on a finite set; we then use functions which return equivalent results on all rotations of a list. On the other hand, we use lists to implement finite sets. We implement set functions as $\cap$, $\times$, $-$ on lists, using an overloaded notation. The result of a function, considered as set is independent of the order of elements in the list.

Lemma $set\ (as \times bs) = set\ as \times set\ bs$

The type $'a\ list$ of all lists of elements of $'a$ is an inductive datatype with constructors [] and #. The base case is the empty list [], and if *xs* is of type

'a list and x is of type *'a*, then $x\#xs$ is again of type *'a list*.
A primitive recursive function f is defined in the following style.

f :: 'a list ⇒ 'b
f [] = ...
f (x#xs) = ... (f xs) ...

Particular lists are denoted like $[a,b,c]$. We concatenate two lists xs and ys by $xs@ys$. Let xs be a list. *hd xs* is the head element of xs; *last xs* is the last element of xs; *butlast xs* is the list xs without the last element; *take n xs* returns the list of the first n elements of the list xs; *drop n xs* returns the list xs without the first n elements; *rev xs* is the reversed list; $|xs|$ is the length of xs; $xs[\![i]\!]$ is the element at position i in xs where $0 \leq i < |xs|$; $xs[i{:=}a]$ is the list xs where the element at position i is replaced by a; *set xs* is the set of elements in xs; $[x \in xs.\ P\ x]$ is the list of all elements of xs that satisfy property P; $[f\ x.\ x \in xs]$ is the list of all elements of $f\ x$ where $x \in$ *set xs*; *replicate n x* is the list consisting of n times the element x; $[0\ ..{<}\ n]$ is the list of numbers from 0 to $n - 1$; $xs \times ys$ a the list of all elements (x,y) where $x \in$ *set xs* and $y \in$ *set ys*; $xs \cap ys$ a the list of all elements x where $x \in$ *set xs* and $x \in$ *set ys* (see Appendix A.1); $xs - ys$ is a list containing all elements x with $x \notin$ *set ys*; $\sum x{\in}xs\ f\ x$ is the sum $\sum_{i=0}^{i<|xs|} f\ xs[\![i]\!]$;
$\cup x \in xs\ f\ x$ is the concatenation of all lists obtained by $f\ x$ where $x \in$ *set xs*.
The function *minimal f x xs* returns the minimal element of the set $\{x\} \cup$ *set xs* with respect to *f*. The functions *minList x ys* and *maxList x ys* calculate the minimal and maximal element set $\{x\} \cup$ *set xs* (see Appendix A.1).

Product Types The type $a' \times 'b$ is the type of all pairs $(a,\ b)$, where a is of type *'a* and b is of type *'b*. A pair is constructed by $(a,\ b)$. We select the first component of a pair $p = (a,b)$ by *fst p* $= a$, and the second component by *snd p* $= b$. We write *'a* $\times$ *'b* $\times$ *'c* for *'a* $\times$ (*'b* $\times$ *'c*).

Arrays We introduce the type *'a array* for lists of fixed length. An array of length n may be constructed by $A = [\![f\ i.\ i < n]\!]$ from elements $f\ i$, where $0 \leq i < n$, and indexed at position i by $A[\![i]\!]$. Higher-dimensional arrays are constructed for example like $[\![f\ x\ y\ z.\ x < n_x,\ y < n_y,\ z < n_z]\!]$ and indexed at position (i,j,k) by $A[\![i,j,k]\!]$.

Option Type The type *'a option* is an inductive datatype with constructors *None* and *Some a* where *a* is of type *'a*. The function *the* applied to an element *Some a* of option type returns *a*.

Tables We introduce a type *('a,'b) table* for all lists of pairs *(a, b)*, where *a* is of type *'a* and *b* is of type *'b* . We introduce functions *removeKeyList* to remove elements from a table, indexed by the first element (see Appendix A.1).

HOL is a logic of total functions. However we may define functions partially; for example, for a function f of type *'a list* $\Rightarrow$ *'b* we may omit a definition for the empty list. Then the result of f [] is an element of type *'b*, however, we can not prove anything about it.

Code generation

For executable functions, Isabelle/HOL provides a mechanism for ML code generation[6], which is based only on equalities proven in Isabelle/HOL. Hence the execution can be trusted not to introduce any inconsistencies. All primitive recursive and well-founded recursive function are translated to recursive ML functions. Finite sets may be implemented by lists. The expressions $\exists\, x \in set\ xs.\ P\ x$ and $\forall\, x \in set\ xs.\ P\ x$ are executable, provided the predicate P is executable. They are implemented by corresponding functions on lists. For acceptable performance natural numbers may be implemented via integers in ML, using appropriate proven equalities for the translation. We also may provide for a function f a more efficient version f', for example by avoiding duplicate calculations, proving equivalence in Isabelle/HOL, and translating f to the ML function generated for f'.

Proofs

The formal proofs in this thesis use the Isabelle/Isar proof language with the aim to generate readable formal proof documents. For an introduction to Isar see [34], here we only present the very basic language elements. In order to proof a lemma "*if A and B then C*", we may assume A and B, prove some intermediate result D, and finally derive C. The proof beginning with "**proof** −" means "do not apply any proof methods".

Lemma $A \Longrightarrow B \Longrightarrow C$

proof −
  assume A **and** B
  have D
  show C
qed

We write "A **then have** B", when we want to use A in the proof of B. We can also name facts **have** a: A. We write "**from** a" to use A in a later proof. We write "C **with** a" to use the previous fact C and A.

  assume a: A
  then have B
  from a **have** C
  with a **have** D

If we start a proof by "**proof**", a standard proof rule is applied. In this example, the proof is split up in two cases, where each case is to be proven separately, separated by "**next**".

Lemma $A \Longrightarrow C$
proof −
  assume A
  then have $A_1 \vee A_2$
  then show C
  proof
    assume A_1 **then show** C
  next
    assume A_2 **then show** C
  qed
qed

If we can derive the existence of an element x with a property $B\ x$, we write "**obtain** x **where** B x" and subsequently prove the existence.

Lemma $A \Longrightarrow C$
proof −
  assume A
  then obtain x **where** $B\ x$
  then show C
qed

We collect facts A_1, A_2, and A_3 we need to prove C by "**moreover**" and make them available in the proof of C by "**ultimately**".

Lemma $A \Longrightarrow C$
proof −
  assume A
  then have A_1
  moreover have A_2
  moreover have A_3
  ultimately show C
qed

We prove a chain of equations (or inequations) by proving every single equation and combining them using transitivity rules. The dots "..." abbreviate the right hand side of the previous equation.

Lemma $a = b$
proof −
  have $a = a_1$
  also have $\ldots = a_2$
  also have $\ldots = a_3$
  also have $\ldots = b$
  finally show $a = b$ **.**
qed

We use the symbol $!$ to indicate a statement not yet formally proved in Isabelle and the symbol $\surd$ for formally proved statements.

Chapter 3

A Formalization of Plane Graphs

In this chapter we present a formalization of plane graphs in Isabelle/HOL based on directed faces. The mathematical foundations were introduced in Section 2.3.

In Section 3.1, we show the general data structure for the representation of graphs, following the representation used in Hales' Java program, but using lists instead of pointer structures. Then, we define the set of plane graphs in Section 3.2 by an inductive set.

3.1 Formalization of Graphs

In this section, we present a formalization of graphs in Isabelle. In Section 3.1.1, we introduce a formalization of faces by lists of vertices and a permutation function. We continue with the definition of equivalence of faces (see Section 3.1.2) and characteristic properties of faces (see Section 3.1.3).

Finally, we define an Isabelle data structure for the representation of graphs (see Section 3.1.5) and introduce some operations on graphs (see Section 3.1.6). This representation is used for an inductive definition of plane graphs in Section 3.2.

3.1.1 Representation of Faces

We formalize faces as permutations of vertices, i.e. we represent them by distinct lists of vertices and provide a function *nextVertex* that yields for every face f the permutation function $\lambda v.\ f{\cdot}v$. Consequently, we have to prove that all operations on faces are preserved under rotation of the vertex list, i.e. that the operations are independent from the actual representation of a face.

We represent vertices by natural numbers.

vertex = *nat*

For the representation of faces, we introduce a new data type, consisting of two components, the (distinct) list of vertices of the face and the type of the face. The type of a face is either final or nonfinal; it is represented as a two-element datatype.

facetype = *Final* | *Nonfinal*
face = *Face* (*vertex list*) *facetype*

We define an overloaded function *final* on faces that determines if a face is final. The function *final* will later also be defined for graphs.

final :: *'a* ⇒ *bool*
final (*Face vs f*) = (*case f of Final* ⇒ *True* | *Nonfinal* ⇒ *False*)

We define selector functions *type* and *vertices* on faces that select the respective component of a face. The function *vertices* is overloaded. Later it will also be defined for graphs. The set of vertices V_f in a face f is denoted by *set* (*vertices f*). A face may be made final by the function *setFinal*.

type :: *'a* ⇒ *facetype*
type (*Face vs f*) = *f*

vertices :: *'a* ⇒ *vertex list*
vertices (*Face vs f*) = *vs*

setFinal :: *face* ⇒ *face*
setFinal f ≡ *Face* (*vertices f*) *Final*

Next Vertex We implement the permutation function of a face by the function *nextVertex* (written as $f{\cdot}v$). This function is based on *nextElem*.

If x is contained in a (distinct) list as and if x is not the last element of as, *nextElem as b x* returns the successor of an element x in as.

Otherwise, the function *nextElem* returns a default element b. $f{\cdot}v$ returns the next vertex for a vertex v in the face f in cyclic order by calling *nextElem* with the head element in the vertex list of f as default element. Hence, *nextVertex* implements the permutation function $f{\cdot}v$.

```
nextElem :: 'a list ⇒ 'a ⇒ 'a ⇒ 'a
nextElem [] b x = b
nextElem (a#as) b x =
  (case as of [] ⇒ b
      | (a'#as') ⇒ if x = a then a' else nextElem as b x)

nextVertex :: face ⇒ vertex ⇒ vertex
f·v ≡ let vs = vertices f in nextElem vs (hd vs) v
```

Note that in Isabelle/HOL all functions need to be total; however, it is assumed that the function *nextVertex* is only applied on vertices v that are contained in the face f and that the vertex list of f is distinct. For the verification this means that we have to prove these preconditions whenever we use properties about *nextVertex*.

The function $f^n{\cdot}v$ returns the n-fold application of $f{\cdot}v$.

```
nextVertices :: face ⇒ nat ⇒ vertex ⇒ vertex
f^n·v ≡ ((f·) ^n) v
```

A vertex v is called *incident* with a face f if v is contained in the set of vertices of f, i.e. $v \in set\ (vertices\ f)$. An *edge* (a, b) is contained in a face f if b is the successor of a in f.

```
edges :: 'a ⇒ (vertex × vertex) list
edges (f::face) ≡ [(a, f·a). a ∈ vertices f]
```

Opposite Face For every face *f*, we obtain the *opposite* (*inverse*) face f^{op} by reversing the cyclic order of the vertices of *f*.

```
(vs::vertex list)^op ≡ rev vs
(Face vs f)^op = Face (rev vs) f
```

We also define the permutation function $f^{-1}{\cdot}v$ of the inverse face f^{op}.

$prevVertex :: face \Rightarrow vertex \Rightarrow vertex$
$f^{-1}{\bullet}v \equiv (let\ vs = vertices\ f\ in\ nextElem\ (rev\ vs)\ (last\ vs)\ v)$

3.1.2 Equivalence of Faces

Two faces f_1 and f_2 are equivalent ($f_1 \cong f_2$) if they represent the same permutation of vertices, i.e. if we can obtain one vertex list from the other by rotation. The type of the faces is insignificant.

The function *rotate n* is a standard Isabelle list function that performs n rotations of a list by 1, where a rotation by 1 of a list $[x_0, x_1, ..., x_n]$ yields $[x_1, ..., x_n, x_0]$. The symbol $\cong$ is further used for the the equivalence relation of faces as a set of pairs of faces:

$vs_1 \cong (vs_2{::}vertex\ list) \equiv \exists n.\ vs_2 = rotate\ n\ vs_1$
$f_1 \cong (f_2{::}face) \equiv vertices\ f_1 \cong vertices\ f_2$

$\cong \equiv \{(f_1, f_2).\ f_1 \cong f_2\}$

3.1.3 Properties of Faces

In order to prove characteristic properties of faces, we introduce the boolean functions *is-sublist* and *is-nextElem xs* on lists.

is-sublist is a relation on lists. *xs* is a sublist of *ys* if we obtain *xs* by cutting off elements at the start or the end of *ys*.

$is\text{-}sublist :: 'a\ list \Rightarrow 'a\ list \Rightarrow bool$
$is\text{-}sublist\ xs\ ys \equiv (\exists as\ bs.\ ys = as\ @\ xs\ @\ bs)$

For a list *xs*, *is-nextElem xs* is the relation of succeeding vertices in *xs* modulo rotation. y is the element of *xs* that follows x modulo rotation. That is, either the list $[x,y]$ is a sublist of *xs* or x is the last element and y the first in *xs*.

$is\text{-}nextElem :: 'a\ list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
$is\text{-}nextElem\ xs\ x\ y \equiv is\text{-}sublist\ [x,y]\ xs \vee xs \neq [] \wedge x = last\ xs \wedge y = hd\ xs$

By using these definitions, we prove several properties of faces: the value returned by the function *nextElem as b x* is either an element of *as* or the default element b. If x is not in the list *as* or x is the last element of *as*, the result is b. If x is an element of the list, the function implements the successor

of x modulo rotation. If two vertex lists as and as' are equivalent, then the functions *nextElem as* (*hd as*) and *nextElem as'* (*hd as'*) are identical on all elements of as (or as'):

Lemma $x \notin set\ as \Longrightarrow nextElem\ as\ b\ x = b$ $\surd$

Lemma $distinct\ as \Longrightarrow nextElem\ as\ b\ (last\ as) = b$ $\surd$

Lemma $distinct\ as \Longrightarrow x \in set\ as \Longrightarrow$
$is\text{-}nextElem\ as\ x\ y = (nextElem\ as\ (hd\ as)\ x = y)$ $\surd$

Lemma $as \cong as' \Longrightarrow distinct\ as \Longrightarrow x \in set\ as \Longrightarrow$
$nextElem\ as\ (hd\ as)\ x = nextElem\ as'\ (hd\ as')\ x$ $\surd$

The permutation function of f applied to an element v of the vertices of f returns an element of the vertices f. $f\cdot v$ is the successor in the vertex list of f modulo rotation. For two equivalent faces f_1 and f_2, the permutation functions are identical on all vertices of f_1 (or f_2):

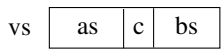
Lemma $v \in set\ (vertices\ f) \Longrightarrow f\cdot v \in set\ (vertices\ f)$ $\surd$

Lemma $distinct\ (vertices\ f) \Longrightarrow v \in set\ (vertices\ f) \Longrightarrow$
$is\text{-}nextElem\ (vertices\ f)\ v\ (f\cdot v)$ $\surd$

Lemma $f_1 \cong f_2 \Longrightarrow distinct\ (vertices\ f_1) \Longrightarrow v \in set\ (vertices\ f_1) \Longrightarrow$
$f_1\cdot v = f_2\cdot v$ $\surd$

3.1.4 Operations on Faces

Splitting a Vertex List at a Vertex First we define some auxiliary functions on lists: The function *splitAt c as* splits a list *as* at an element c in two lists: the first part contains all elements from the first element of *as* up to (and excluding) c, the second one contains all elements from (excluding) c up to the last element of the list. If c is not contained in *as*, the function returns the original list *as* and the empty list:



$splitAtRec :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list \times 'a\ list$

$splitAtRec\ c\ bs\ [] = (bs,[])$
$splitAtRec\ c\ bs\ (a\#as) = (if\ a = c\ then\ (bs,\ as)\ else\ splitAtRec\ c\ (bs@[a])\ as)$

$splitAt :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list \times 'a\ list$
$splitAt\ c\ as \equiv splitAtRec\ c\ []\ as$

The function *splitAt ram vs* is characterized by the following properties. If *ram* is none of the vertices of *vs* then the function returns the pair $(vs, [])$. If *ram* is one of the vertices of *vs* then *splitAt* returns a pair $(as,\ bs)$ where $vs = as\ @\ [ram]\ @\ bs$.

Lemma $ram \notin set\ vs \Longrightarrow splitAt\ ram\ vs = (vs, [])$ √

Lemma $ram \in set\ vs \Longrightarrow (as,\ bs) = splitAt\ ram\ vs \Longrightarrow$
$vs = as\ @\ [ram]\ @\ bs$ √

Normalized Faces For the verification, we introduce a function *verticesFrom f v*, which rotates the vertex list of a face f such that v is the head element. The rotated list is equivalent to the original vertex list and contains at position i the ith successor of v in f. This function is not used for the execution of the algorithm, but only for the verification.

$verticesFrom :: face \Rightarrow vertex \Rightarrow vertex\ list$
$verticesFrom\ f\ v \equiv v\ \#\ snd\ (splitAt\ v\ (vertices\ f))\ @\ fst\ (splitAt\ v\ (vertices\ f))$

Lemma $distinct\ (vertices\ f) \Longrightarrow v \in set\ (vertices\ f) \Longrightarrow$
$vertices\ f \cong verticesFrom\ f\ v$ √

Lemma $distinct\ (vertices\ f) \Longrightarrow i < |vertices\ f| \Longrightarrow$
$v \in set\ (vertices\ f) \Longrightarrow (verticesFrom\ f\ v)[\![i]\!] = f^i \cdot v$ √

Moreover, we can normalize faces such that we can reduce the equivalence test of faces to a comparison of the normalized vertex lists. For the definition of *minList* see Appendix A.1.

$minVertex :: face \Rightarrow vertex$
$minVertex\ f \equiv minList\ (hd\ (vertices\ f))\ (vertices\ f)$

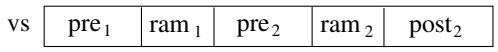
$normFace :: face \Rightarrow vertex\ list$
$normFace\ f \equiv verticesFrom\ f\ (minVertex\ f)$

Lemma $vertices\ f_1 \neq [] \Longrightarrow (f_1 \cong f_2) = (normFace\ f_1 = normFace\ f_2)$ √

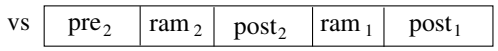
Selecting the Vertices Between two Vertices The function *between vs* ram_1 ram_2 returns a list that contains all elements of *vs* between ram_1 and ram_2 (both end points excluded), considering *vs* as a ring. The precondition for this function is that *vs* is distinct, both ram_1 and ram_2 occur in *vs*, and $ram_1 \neq ram_2$. We split *vs* at ram_1 in two lists pre_1 and $post_1$, then we split up the sublist that contains ram_2 in two lists pre_2 and $post_2$.

There are two cases to distinguish:

1. ram_1 appears before ram_2 in *vs*, i.e. ram_2 occurs in $post_1$. Then the result is pre_2.



2. ram_2 appears before ram_1 in *vs* or $c = b$, i.e. ram_2 does not occur in $post_1$. Then the result is $post_1$ @ pre_2.



between :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *'a list*
between vs ram_1 ram_2 ≡
  let (pre_1, $post_1$) = *splitAt* ram_1 *vs in*
  if ram_2 ∈ *set* $post_1$
  then let (pre_2, $post_2$) = *splitAt* ram_2 $post_1$ *in* pre_2
  else let (pre_2, $post_2$) = *splitAt* ram_2 pre_1 *in* $post_1$ @ pre_2

We summarize the preconditions for the function *between vs* ram_1 ram_2 in a predicate *pre-between*:

pre-between :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *bool*
pre-between vs ram_1 ram_2 ≡
  distinct vs ∧ ram_1 ∈ *set vs* ∧ ram_2 ∈ *set vs* ∧ $ram_1 \neq ram_2$

We also introduce a list function *before*. *before vs* ram_1 ram_2 determines whether an element ram_1 occurs before an element ram_2 in *vs*.

before :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *bool*
before vs ram_1 ram_2 ≡ ∃ *as bs cs*. *vs* = *as* @ ram_1 # *bs* @ ram_2 # *cs*

Based on this definitions we prove characteristic properties of the function *between*: Under the assumption that the precondition of *between* holds, and

if we split *vs* at ram_1 in (*as*, *bs*) and the condition of the if-statement $ram_2 \in set\ bs$ holds, then ram_1 occurs before ram_2 in *vs* and *between vs* ram_1 ram_2 contains the elements of *vs* between ram_1 and ram_2 in *vs*. Otherwise ram_2 occurs before ram_1 and the function returns the list of all elements from ram_1 to the end of *vs* and then from the beginning of *vs* to ram_2. The result of *between vs* ram_1 ram_2 is independent of the actual representation of *vs*, i.e. rotation does not change the result:

Lemma *pre-between vs* ram_1 $ram_2 \Longrightarrow$
$(\neg\ before\ vs\ ram_1\ ram_2) = before\ vs\ ram_2\ ram_1$ √

Lemma *distinct vs* $\Longrightarrow$ (*as*, *bs*) = (*splitAt* ram_1 *vs*) $\Longrightarrow$
$ram_2 \in set\ bs = before\ vs\ ram_1\ ram_2$ √

Lemma *pre-between vs* ram_1 $ram_2 \Longrightarrow$ *before vs* ram_1 $ram_2 \Longrightarrow$
$\exists\, as\ bs.\ vs = as\ @\ [ram_1]\ @\ between\ vs\ ram_1\ ram_2\ @\ [ram_2]\ @\ bs$ √

Lemma *pre-between vs* ram_1 $ram_2 \Longrightarrow$ *before vs* ram_2 $ram_1 \Longrightarrow$
$\exists\, as\ bs\ cs.\ between\ vs\ ram_1\ ram_2 = cs\ @\ as$
$\wedge\ vs = as\ @\ [ram_2]\ @\ bs@\ [ram_1]\ @\ cs$ √

Lemma *between-eqF*: *pre-between vs* ram_1 $ram_2 \Longrightarrow$ *eqF vs* $vs' \Longrightarrow$
between vs ram_1 ram_2 = *between* vs' *ram1 ram2* √

Directed Length The function *directedLength f a b* counts the number of edges in the face f between two vertices a and b.

directedLength :: *face* ⇒ *vertex* ⇒ *vertex* ⇒ *nat*
directedLength f a b ≡ *if a* = *b then 0 else* |*between* (*vertices f*) *a b*| + *1*

3.1.5 Representation of Graphs

We formalize graphs as sets of faces. In order to make all definitions executable, a set A is represented by a distinct list containing all elements of A. Hence a graph is represented by a distinct list of faces. Consequently, all graph operations and properties must be independent of the order in which the faces are stored in the list.

A graph g is represented as a data type with the following components:

- *faces* g, the list of faces in g.

- *countVertices* g, the number of vertices in g.
- *faceListAt* g, an incidence list of face lists, associating to each vertex v a list of faces in g containing v. The function *facesAt* g assigns to every vertex v in g the list of faces incident with v, i.e. the element at position v in the list *faceListAt* g.
- *heights* g, a list of natural numbers, associating to each vertex a height. The function *height* g assigns to a vertex v the element at position v in the list *heights* g. This is only used for optimization purposes.
- *baseVertex* g, one of the vertices.

The list of vertices of a graph contains the natural numbers from 0 upto (excluding) *countVertices*.

Additionally, we define the corresponding selector functions on graphs.

$graph = Graph\ (face\ list)\ nat\ face\ list\ list\ nat\ list\ (vertex\ option)$

$faces :: graph \Rightarrow face\ list$
$faces\ (Graph\ fs\ n\ f\ h\ b) = fs$

$countVertices :: graph \Rightarrow nat$
$countVertices\ (Graph\ fs\ n\ f\ h\ b) = n$

$vertices\ (Graph\ fs\ n\ f\ h\ b) = [0\ ..<\ n]$

$faceListAt :: graph \Rightarrow face\ list\ list$
$faceListAt\ (Graph\ fs\ n\ f\ h\ b) = f$

$facesAt :: graph \Rightarrow vertex \Rightarrow face\ list$
$facesAt\ g\ v \equiv if\ v \in set(vertices\ g)\ then\ faceListAt\ g\ [\![v]\!]\ else\ []$

$heights :: graph \Rightarrow nat\ list$
$heights\ (Graph\ fs\ n\ f\ h\ b) = h$

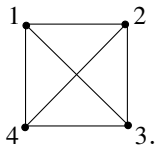
$height :: graph \Rightarrow vertex \Rightarrow nat$
$height\ g\ v \equiv heights\ g\ [\![v]\!]$

$baseVertex :: graph \Rightarrow vertex\ option$
$baseVertex\ (Graph\ fs\ n\ f\ h\ b) = b$

Note that the representation is highly redundant in order to avoid duplicate calculations and obtain acceptable performance for the enumeration algorithm. For example, *countVertices g* and *faceListAt g* could be calculated from *faces g*. The last two components are only used for optimization of the generation process. The general definition also allows graphs that are not *well-formed* (*inconsistent*): for example, if the incidence list has not the same length as the number of vertices. Well-formedness is guaranteed for the set of plane graphs by its inductive definition (see Section 3.2). This can be proved by induction on the construction. Note that the definition of plane graphs (see Section 3.2) imposes further restrictions on graphs (see Section 2.3).

The general definition allows graphs that do not correspond to graphs drawn in the plane, like in the following example:

Example
The following graph represented by the set of faces {[*1*,*2*,*3*], [*1*,*3*,*4*], [*1*,*2*,*4*], [*2*,*3*,*4*], [*4*,*3*,*2*,*1*]} is not plane.



3.1.6 Operations on Graphs

A graph is *final*, iff all faces are final:

finals :: *graph* ⇒ *face list*
finals g ≡ [*f* ∈ *faces g*. *final f*]

nonFinals :: *graph* ⇒ *face list*
nonFinals g ≡ [*f* ∈ *faces g*. ¬ *final f*]

final g ≡ (*nonFinals g* = [])

A vertex is *final* if all incident faces are final; i.e. there are no non-final incident faces:

nonFinalsAt :: *graph* ⇒ *vertex* ⇒ *face list*
nonFinalsAt g v ≡ [*f* ∈ *facesAt g v*. ¬ *final f*]

$finalVertex :: graph \Rightarrow vertex \Rightarrow bool$
$finalVertex\ g\ v \equiv (nonFinalsAt\ g\ v = [])$

The *degree* of a vertex v in a graph g is the number of faces incident with v in g. *tri g v*is the number of incident final triangles, *quad g v* is the number of incident final quadrilaterals, and *except g v* is the number of incident final exceptional faces:

$degree :: graph \Rightarrow vertex \Rightarrow nat$
$degree\ g\ v \equiv |facesAt\ g\ v|$

$tri :: graph \Rightarrow vertex \Rightarrow nat$
$tri\ g\ v \equiv |[f \in facesAt\ g\ v.\ final\ f \wedge |vertices\ f| = 3]|$

$quad :: graph \Rightarrow vertex \Rightarrow nat$
$quad\ g\ v \equiv |[f \in facesAt\ g\ v.\ final\ f \wedge |vertices\ f| = 4]|$

$except :: graph \Rightarrow vertex \Rightarrow nat$
$except\ g\ v \equiv |[f \in facesAt\ g\ v.\ final\ f \wedge 5 \leq |vertices\ f|\ ]|$

An *edge* (a, b) is contained in a graph g if it is contained in some face f of g.

$edges\ (g{::}graph) \equiv \cup_{f\ \in\ faces\ g}\ edges\ f$

The function *neighbors* calculates all vertices adjacent to a vertex v.

$neighbors :: graph \Rightarrow vertex \Rightarrow vertex\ list$
$neighbors\ g\ v \equiv [f{\cdot}v.\ f \in facesAt\ g\ v]$

The function *nextFace*, denoted by $(g,v){\cdot}f$, permutes the faces at a vertex v. *prevFace*, denoted by $(g,v)^{-1}{\cdot}f$, is the inverse operation.

$nextFace :: graph \times vertex \Rightarrow face \Rightarrow face$
$(g,v){\cdot}f \equiv (let\ fs = (facesAt\ g\ v)\ in$
$\quad (case\ fs\ of\ [] \Rightarrow f$
$\qquad | g\#gs \Rightarrow nextElem\ fs\ (hd\ fs)\ f))$

$prevFace :: graph \times vertex \Rightarrow face \Rightarrow face$
$(g,v)^{-1}{\cdot}f \equiv (let\ fs = (facesAt\ g\ v)\ in$
$\quad (case\ fs\ of\ [] \Rightarrow f$
$\qquad | g\#gs \Rightarrow nextElem\ (rev\ fs)\ (last\ fs)\ f))$

3.2 Formalization of Plane Graphs

We inductively define (connected) plane graphs.

We start the construction with *initial* graphs (*seed graphs*). Initial graphs consist one final face and one nonfinal outer face, where the final one is the inverse face of the nonfinal one (see Section 3.2.1).

The basic graph modifications are splitting a nonfinal face in two faces, and making a nonfinal face final. The face split operation is explained in Section 3.2.2, the application to a graph in Section 3.2.3. The operation of making a nonfinal face final is described in Section 3.2.4.

One step in the construction of a plane graph g consists of adding a new final face in a nonfinal face f of g by means of these two basic operations (see Section 3.2.4). We say we apply a *patch* for f at an edge e of f. A patch p is represented by a graph with at least 2 faces, one final face f_n (the final face we want to add to the graph), one nonfinal face f' (the inverse face of f, i.e. the boundary of the patch) and 0 or more nonfinal faces, uniquely determined by f and f_n, filling the 'gap' between the old nonfinal face f and the new final face f_n.

We apply a patch by replacing f in g by $p - \{f'\}$. The new nonfinal faces complete the new graph such that, again, every edge is contained in exactly two faces in opposite directions. Hence, this property of plane graphs is preserved. In Section 3.2.5 we explain in detail the enumeration of all possible patches.

Example
All possible patches at a fixed edge for a face of length 6 with a final face of length 3 are presented in Figure 3.1.

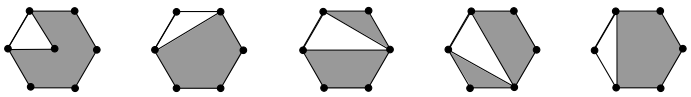


Figure 3.1: Patches of length 3

Definition
We define a partial plane graph inductively by the following two clauses:

(initial) Every initial graph is a partial plane graph.

(add face) Given a partial plane graph, the graph obtained by adding one final face is a partial plane graph.

A *final graph* is one in which every face is final. Only final plane graphs are considered as plane graphs.

We present the Isabelle formalization of this inductive definition of a tree of generated graphs in Section 3.2.6.

Every graph generated using this inductive definition is plane. This can be verified by induction over the generation, using Euler's formula. Furthermore it must be shown that we can reach every plane graph using this construction, i.e. that our definition actually covers all plane graphs.

Theorem
Every plane graph can be reached using the construction described above.
Proof
Let g be any plane graph. We can choose any face as the initial graph. Let h be a nonempty connected subset of g and h' a partial plane graph, reachable by the inductive definition, such that the set of faces of h is the set of final faces of h'. Then g can be reached from h'. The proof is by induction on the cardinality of the difference of set of faces in g and the set of final faces in h'. We show that we can always add a new final face to h': if h' is nonempty, there is a nonfinal face f in h' that shares an edge with one of the final faces in h'. In g there is exactly one face f_2 that shares this edge with one of the final faces in h. We can add the final face f_2 to h' with a patch for the nonfinal face f with final face f_2. The theorem follows by the induction hypothesis.
Qed

We refine the process of generating graphs by successively generating graphs with maximum face length n, starting from $n = 3, 4, \ldots$ (see Figure 3.2). We can get the set of all plane graphs with maximum face length n by starting with an initial graph with face length n, and in every step adding only new faces with length between 3 and n. Note that the sets of graphs generated by these trees are distinct.

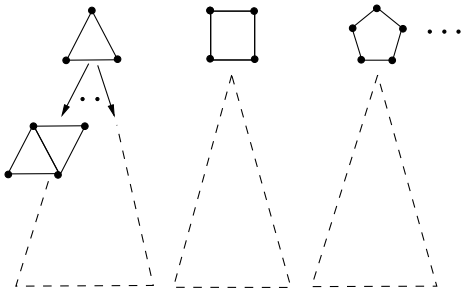


Figure 3.2: Trees of generated plane graphs

3.2.1 Construction of Seed Graphs

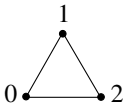
An *initial graph* (*simple seed graph*) is a graph with one final face of length n and one nonfinal face of length n (in opposite direction).

It is constructed by the function *graph*.

```
graph :: nat ⇒ graph
graph n ≡
    (let vs = [0 ..< n];
    fs = [Face vs Final, Face (rev vs) Nonfinal];
    b = (if n < 5 then None else Some 0)
    in (Graph fs n (replicate n fs) (replicate n 0) b))
```

Example
An initial triangle graph has one final face containing the vertices $[0, 1, 2]$ and one nonfinal face containing the vertices $[2, 1, 0]$. The set of vertices is $[0, 1, 2]$ and for each vertex in the graph, the list of adjacent faces contains both faces. The height of a vertex v is the minimal distance of v to one of the vertices of the seed graph, hence the values for the heights of the vertices are initially 0.

```
graph 3 =
Graph [Face [0, 1, 2] Final, Face [2, 1, 0] Nonfinal]
      3
      [[Face [0, 1, 2] Final, Face [2, 1, 0] Nonfinal],
       [Face [0, 1, 2] Final, Face [2, 1, 0] Nonfinal],
       [Face [0, 1, 2] Final, Face [2, 1, 0] Nonfinal]]
      [0, 0, 0]
      None
```



3.2.2 Splitting a Nonfinal Face

The basic operation for the modification of graphs is to *split* a face, implemented by the function *splitFace*. *splitFace* splits a (nonfinal) face f at two vertices ram_1 and ram_2, inserting a list of new vertices ($newVs$) in the interior of f and producing a pair (f_1, f_2) of new nonfinal faces (see Figure 3.3). f_1 contains the vertices of f from ram_1 to ram_2 (including the end points ram_1 and ram_2) and the new vertices $newVs$. f_2 contains the vertices of f from ram_2 to ram_1 and the new vertices $newVs$ in reversed order.

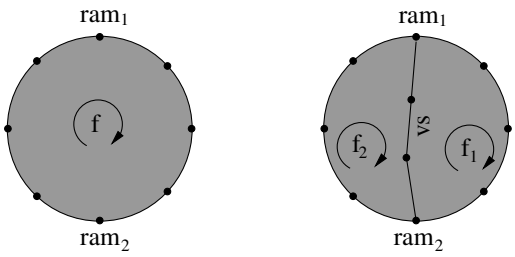


Figure 3.3: Split Face Operation

$splitFace :: face \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex\ list \Rightarrow face \times face$
$splitFace\ f\ ram_1\ ram_2\ newVs \equiv let\ vs = vertices\ f;$
$f_1 = [ram_1] \mathbin{@} between\ vs\ ram_1\ ram_2 \mathbin{@} [ram_2];$
$f_2 = [ram_2] \mathbin{@} between\ vs\ ram_2\ ram_1 \mathbin{@} [ram_1]\ in$
$(Face\ (newVs \mathbin{@} f_1)\ Nonfinal,$
$Face\ (f_2 \mathbin{@} rev\ newVs)\ Nonfinal)$

We define the precondition of the function *splitFace* by a predicate *pre-splitFace*. Both vertices ram_1 and ram_2 must be elements of the vertex list of $oldF$ and the set of vertices in $newVs$ must be new in $oldF$:

$pre\text{-}splitFace :: face \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex\ list \Rightarrow bool$
$pre\text{-}splitFace\ oldF\ ram_1\ ram_2\ newVs \equiv distinct\ (vertices\ oldF) \wedge distinct\ newVs$
$\wedge\ (set\ (vertices\ oldF) \cap (set\ newVs)) = \{\} \wedge ram_1 \neq ram_2$
$\wedge\ ram_1 \in set\ (vertices\ oldF) \wedge ram_2 \in set\ (vertices\ oldF)$

The new faces preserve the successor relation of $oldF$. Every edge (a,b) of $oldF$ is contained in exactly one of the new faces.

Lemma $(f_1, f_2) = splitFace\ oldF\ ram_1\ ram_2\ newVs \Longrightarrow$
$pre\text{-}splitFace\ oldF\ ram_1\ ram_2\ newVs \Longrightarrow (a, b) \in set\ (edges\ oldF) \Longrightarrow$
$(a,b) \in set\ (edges\ f_1) = (\neg\ (a,b) \in set\ (edges\ f_2))$ *!*

Lemma $(f_1, f_2) = splitFace\ oldF\ ram_1\ ram_2\ newVs \Longrightarrow$
$pre\text{-}splitFace\ oldF\ ram_1\ ram_2\ newVs \Longrightarrow (a,b) \in set\ (edges\ f_1) \Longrightarrow$
$(a,b) \in set\ (edges\ oldF) \vee is\text{-}sublist\ [x,y]\ ([ram_2] @ newVs @ [ram_1])$ *!*

Lemma $(f_1, f_2) = splitFace\ oldF\ ram_1\ ram_2\ newVs \Longrightarrow$
$pre\text{-}splitFace\ oldF\ ram_1\ ram_2\ newVs \Longrightarrow (a,b) \in set\ (edges\ f_2) \Longrightarrow$
$(a,b) \in set\ (edges\ oldF) \vee is\text{-}sublist\ [x,y]\ ([ram_1] @ rev\ newVs @ [ram_2])$ *!*

3.2.3 Splitting a Nonfinal Face in a Graph

Replacing Faces We introduce an auxiliary function *replacefacesAt* that modifies the face incidence list F, replacing a face f by a list of faces fs at all positions $F[\![i]\!]$, where $i \in ns$.

This function *replacefacesAt* is based on *replace*(see Appendix A.1).

$replacefacesAt :: nat\ list \Rightarrow face \Rightarrow face\ list \Rightarrow face\ list\ list \Rightarrow face\ list\ list$
$replacefacesAt\ ns\ f\ fs\ F \equiv mapAt\ ns\ (replace\ f\ fs)\ F$

Heights of New Vertices The height of a vertex v is the minimal distance of v to one of the vertices of the seed graph. This is only used for optimization purposes. When we add a new face in a nonfinal face f, we are free to choose a vertex where we add the new face. We add new faces at a vertex with minimal height in order to obtain more 'compact' graphs, which allow to calculate better lower bounds and to detect graphs earlier that will not produce any tame graphs:

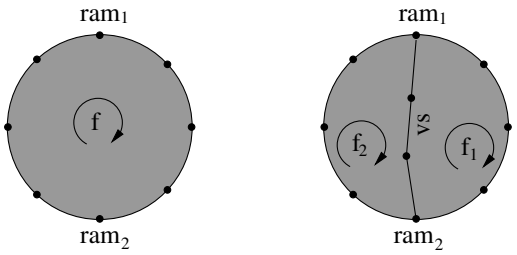
$heightsNewVertices :: nat \Rightarrow nat \Rightarrow nat \Rightarrow nat\ list$
$heightsNewVertices\ h_1\ h_2\ n \equiv [min\ (h_1 + i + 1)\ (h_2 + n - i).\ i \in [0\ ..{<}\ n]]$

Example
$heightsNewVertices\ 0\ 2\ 4 = [1, 2, 3, 3]$

FaceDivisionGraph Now we define a function $FaceDivisionGraph\ g\ ram_1\ ram_2\ oldF\ newVs$, which splits a nonfinal face $oldF$ in a graph g by applying a *splitFace* operation on the face $oldF$. This introduces a set of new vertices $newVs$ in g and replaces $oldF$ by two new nonfinal faces f_1 and f_2. Hence the following modifications must be applied to the graph g. The set of vertices of g is augmented by the list of new vertices $newVs$. In the set of faces of g,

the nonfinal face *oldF* is replaced by the new faces f_1 and f_2. The number of vertices is increased by the number of new vertices. The list *faceListAt g v* of incident faces is changed depending on v:

- For all vertices v between ram_1 and ram_2, *oldF* is replaced by f_1.
- For all vertices v between ram_2 and ram_1, *oldF* is replaced by f_2.
- For $v = ram_1$ and for $v = ram_2$, *oldF* is replaced by f_1 and f_2.
- For all new vertices v, *faceListAt g v* initially contains only f_1 and f_2.




```
FaceDivisionGraph
  :: graph ⇒ vertex ⇒ vertex ⇒ face ⇒ vertex list ⇒ face × face × graph
FaceDivisionGraph g ram1 ram2 oldF newVs ≡
  let fs = faces g;
  n = countVertices g;
  Fs = faceListAt g;
  h = heights g;
  b = baseVertex g;
  vs1 = between (vertices oldF) ram1 ram2;
  vs2 = between (vertices oldF) ram2 ram1;
  (f1, f2) = splitFace oldF ram1 ram2 newVs;
  Fs = replacefacesAt vs1 oldF [f1] Fs;
  Fs = replacefacesAt vs2 oldF [f2] Fs;
  Fs = replacefacesAt [ram1] oldF [f2, f1] Fs;
  Fs = replacefacesAt [ram2] oldF [f1, f2] Fs;
  Fs = Fs @ replicate |newVs| [f1, f2] in
  (f1, f2, Graph ((replace oldF [f2] fs)@ [f1])
                 (n + |newVs| )
                 Fs
                 (h @ heightsNewVertices h⟦ram1⟧ h⟦ram2⟧ |newVs| )
                 b)
```

We define the precondition of the function *FaceDivisionGraph* by a predicate *pre-FaceDivisionGraph*.

pre-FaceDivisionGraph :: *graph* ⇒ *vertex* ⇒ *vertex* ⇒ *face* ⇒ *vertex list* ⇒ *bool*
pre-FaceDivisionGraph g ram$_1$ *ram*$_2$ *oldF newVs* ≡
oldF ∈ *set* (*faces g*) ∧ *distinct* (*vertices oldF*) ∧ *distinct newVs*
∧ *set* (*vertices g*) ∩ *set newVs* = {}
∧ *set* (*vertices oldF*) ∩ *set newVs* = {}
∧ *ram*$_1$ ∈ *set* (*vertices oldF*) ∧ *ram*$_2$ ∈ *set* (*vertices oldF*) ∧ *ram*$_1$ ≠ *ram*$_2$
∧ ((*ram*$_1$,*ram*$_2$) ∉ *set* (*edges oldF*) ∧ (*ram*$_2$,*ram*$_1$) ∉ *set* (*edges oldF*)
∧ (*ram*$_1$,*ram*$_2$) ∉ *set* (*edges g*) ∧ (*ram*$_1$,*ram*$_2$) ∉ *set* (*edges g*) ∨ *newVs* ≠ [])

FaceDivisionGraph replaces a face *oldF* of a graph *g* by two new faces f_1 and f_2 obtained by a *splitFace* operation:

Lemma *oldF* ∈ *set* (*faces g*) ⟹
(f_1, f_2, *newGraph*) = *FaceDivisionGraph g ram*$_1$ *ram*$_2$ *oldF newVs* ⟹
set (*faces newGraph*) = {f_1, f_2} ∪ *set* (*faces g*) − {*oldF*} *!*

Lemma *oldF* ∈ *set* (*faces g*) ⟹
(f_1, f_2, *newGraph*) = *FaceDivisionGraph g ram*$_1$ *ram*$_2$ *oldF newVs* ⟹
(f_1, f_2) = *splitFace oldF ram*$_1$ *ram*$_2$ *newVs* √

3.2.4 Adding a New Final Face in a Nonfinal Face

There are two ways of adding a new final face to a nonfinal graph *g*:

1. Replace a nonfinal face f of g by a final one.
2. Select a nonfinal face f and an edge e in f. Replace f by a *patch* for f. i.e. one new final face and possibly some nonfinal faces are introduced, such that the new face has at least one edge in common with f and contains only vertices of f or new vertices.

Replacing a Nonfinal Face by a Final One

We replace a nonfinal face f of a graph g by a final face using the function *makeFaceFinal*, which replaces all occurrences of f in the face list and all incidence lists by the final copy of f.

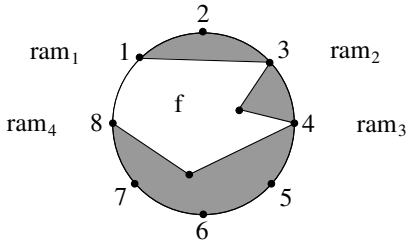
The function *makeFaceFinal* is based on a list function *replace* (see Appendix A.1).

```
makeFaceFinalFaceList :: face ⇒ face list ⇒ face list
makeFaceFinalFaceList f fs ≡ replace f [setFinal f] fs

makeFaceFinal :: face ⇒ graph ⇒ graph
makeFaceFinal f g ≡
    Graph (makeFaceFinalFaceList f (faces g))
          (countVertices g)
          [makeFaceFinalFaceList f fs. fs ∈ faceListAt g]
          (heights g)
          (baseVertex g)
```

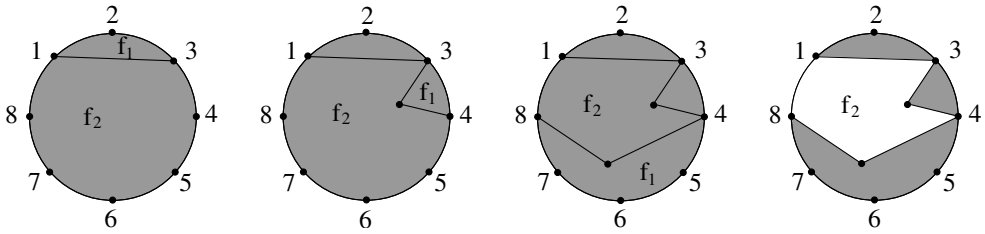
Adding a New Final Face in a Nonfinal Face

The function *addFace g f vs* corresponds to applying a patch for f (see Section 3.2, **add face**). It adds a new final face to a graph g in a nonfinal face f of g. The vertices of the new face are given by the list *vs*: if an element $v \in$ *set vs* is *None*, a new vertex is produced if it is *Some w*, where $w \in$ *set vertices f*, then w will be the next vertex in the new final face. Hence the *FaceDivisionGraph* operation is applied at vertex w.

Example
An *addFace* operation with vertex list of the new final face *vs* = [*Some 1*, *Some 3*, *None*, *Some 4*, *None*, *Some 8*].



The function *addFace g f vs* searches for the first vertex w_1 in the vertex list *vs*, then *addFaceSnd g f* w_1 *n vs* searches for the next vertex w_2 in *vs*, starting at the given vertex w_1. For every *None* element between w_1 and w_2 a new vertex is created, the number of new vertices is counted by the variable n. If the vertices w_1 and w_2 are neighbors and there are no new vertices between w_1 and w_2, then there is nothing to do. Otherwise a face split is applied between w_1 and w_2 with n new vertices. Then the function continues with the remaining vertices of *vs*, starting at the vertex w_2 in the new nonfinal face f_2, the face containing all vertices of f between w_2 and w_1. Finally, when *vs* is empty, the last nonfinal face f_2 is made final.



For the special case in which the vertex list *vs* of the new face contains all vertices of the old face, starting at v, the nonfinal face f is made final, and no new nonfinal faces are created.

```
addFaceSnd :: graph ⇒ face ⇒ vertex ⇒ nat ⇒ vertex option list ⇒ graph
addFaceSnd g f w1 n [] = makeFaceFinal f g
addFaceSnd g f w1 n (v#vs) =
  (case v of None ⇒ addFaceSnd g f w1 (Suc n) vs
    | (Some w2) ⇒
      if f·w1 = w2 ∧ n = 0
      then addFaceSnd g f w2 0 vs
      else let ws = [countVertices g  ..< countVertices g + n];
      (f1, f2, g') = FaceDivisionGraph g w1 w2 f (rev ws) in
      addFaceSnd g' f2 w2 0 vs)

addFace :: graph ⇒ face ⇒ vertex option list ⇒ graph
addFace g f [] = g
addFace g f (v#vs) =
  (* search for starting point: vertex followed by null or a non-adjacent edge *)
  (case v of None ⇒ addFace g f vs
     | (Some w1) ⇒ addFaceSnd g f w1 0 vs)
```

The predicate *pre-addFace* summarizes the preconditions for the *addFace operation*. The function *removeNones* filters out all *None* elements of a list.

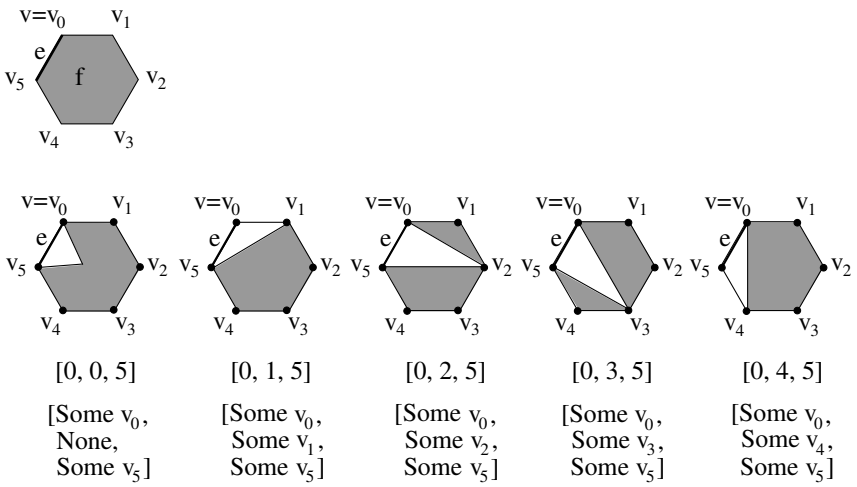
```
removeNones :: 'a option list ⇒ 'a list
removeNones vs ≡ [the v. v ∈ vs, (λv. v ≠ None)]

pre-addFace :: face ⇒ vertex option list ⇒ bool
pre-addFace f vs ≡
    let w = the (hd vs) in
       [v ∈ verticesFrom f w. v ∈ set (removeNones vs)] = removeNones vs
  ∧ distinct (vertices f)
  ∧ w ∈ set (vertices f)
  ∧ last vs =  Some (last (verticesFrom f w))
  ∧ 2 < | vs |
```

3.2.5 Enumeration of all Possible Patches

Now we aim for a function that completely enumerates all possible patches. Given a nonfinal face f of length n in a graph g and a vertex v in f. Let e be the edge in f which ends in v.

We calculate the set of successor graphs (represented as list) obtained by applying a patch for f, such that the new final face f_2 has the edge e in common with f.

Example
All patches for a face of length 6 with a new final triangle:



Let m be the length of the new final face f_2 in a patch.

Assume that the vertices in f are labeled by $v_0, v_1, \ldots, v_{n-1}$ starting from $v_0 = v$, such that f_2 contains the edge from v_{n-1} to v_0.

Every patch can be represented by an increasing list of indices $[i_0, ..., i_{m-1}]$ of length m with first element $i_0 = 0$, last element $i_{m-1} = n-1$ and penultimate element $i_{m-2} < i_{m-1}$, i.e.

$$0 = i_0 \leq i_1 \leq \ldots \leq i_{m-2} < i_{m-1} = n - 1.$$

The list of all possible patches is calculated in three steps:

1. *enumerator m n* calculates the index lists for all patches for a nonfinal face of length n with a new final face of length m.

2. *indexToVertexList v f is* converts an index list to a list of optional vertices, where *Some* v_i represents an existing vertex v_i in the face f and *None* represents a new vertex.

3. *addFace g f vs* constructs a new graph by applying the patch to the face f in g.

Enumeration of Increasing Index Lists

The function *enumerator* produces all increasing lists of length m with first element 0, last element $n-1$ and penultimate element at most $n-2$. The first element in every list is 0 and the last element is $n-1$. In between there is an increasing list of $m-2$ elements of the set $\{0, .., n-2\}$. The function *enumBase* creates all singleton lists with an element of the set $\{0, .., nmax\}$, where $nmax = n-2$. The function *enumAppend* extends every list *is* by one element that is at least the last element of *is* and at most the maximal value *nmax*:

enumBase :: *nat* ⇒ *nat list list*
enumBase nmax ≡ [[*i*]. *i* ∈ [*0* .. *nmax*]]

enumAppend :: *nat* ⇒ *nat list list* ⇒ *nat list list*
enumAppend nmax iss ≡ $\cup_{is \in iss}$ [*is* @ [*n*]. *n* ∈ [*last is* .. *nmax*]]

enumerator :: *nat* ⇒ *nat* ⇒ *nat list list*
enumerator inner outer ≡ *if inner* < *3 then* [[*0*, *outer* − *1*]]
else let nmax = *outer* − *2*; *k* = *inner* − *3 in*
[[*0*] @ *is* @ [*outer* − *1*]. *is* ∈ ((*enumAppend nmax*) ^ *k*) (*enumBase nmax*)]

Example

The index lists for all patches for a face of length 6 with a new triangle are *enumerator 3 6* = [[*0*, *0*, *5*], [*0*, *1*, *5*], [*0*, *2*, *5*], [*0*, *3*, *5*], [*0*, *4*, *5*]]

For a new quadrilateral we have *enumerator 4 6* =[[*0*, *0*, *0*, *5*], [*0*, *0*, *1*, *5*], [*0*, *0*, *2*, *5*], [*0*, *0*, *3*, *5*], [*0*, *0*, *4*, *5*], [*0*, *1*, *1*, *5*], [*0*, *1*, *2*, *5*], [*0*, *1*, *3*, *5*], [*0*, *1*, *4*, *5*], [*0*, *2*, *2*, *5*], [*0*, *2*, *3*, *5*], [*0*, *2*, *4*, *5*], [*0*, *3*, *3*, *5*], [*0*, *3*, *4*, *5*], [*0*, *4*, *4*, *5*]].

The predicate *increasing* defines the property of a list to be increasing: the successor y of an element x in the list is at least x. The predicate *incrIndexList* characterizes the index lists that represent patches. A patch of a nonfinal face of length n with a new final face of length m is represented by an increasing list of length m with head element 0, last element $n-1$ and penultimate element smaller than $n-1$.

$increasing :: ('a :: linorder)\ list \Rightarrow bool$
$increasing\ ls \equiv \forall x\ y\ as\ bs.\ ls = as\ @\ x\ \#\ y\ \#\ bs \longrightarrow x \leq y$

$incrIndexList :: nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow bool$
$incrIndexList\ ls\ m\ n \equiv$
$\quad 1 < m \wedge 1 < n \wedge hd\ ls = 0 \wedge last\ ls = (n - 1) \wedge |ls| = m$
$\quad \wedge\ last\ (butlast\ ls) < last\ ls \wedge increasing\ ls$

We prove both correctness and completeness of the enumeration function.

Lemma $1 < m \Longrightarrow 1 < n \Longrightarrow$
$ls \in set\ (enumerator\ m\ n) \Longrightarrow incrIndexList\ ls\ m\ n$ $\surd$

Lemma $incrIndexList\ ls\ m\ n \Longrightarrow ls \in set\ (enumerator\ m\ n)$ $\surd$

Conversion to a List of Vertices

indexToVertexList converts a list of indices to a list of vertices in f starting at a vertex v in f. Now, for every element v in the list which is the same as its predecessor in the list, a new vertex must be created. This is indicated by replacing v by *None*. Otherwise v remains in the list as *Some v*. We calculate the list of vertices from the index list, and replace duplicate vertices by *None*:

$hideDupsRec :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ option\ list$
$hideDupsRec\ a\ [] = []$
$hideDupsRec\ a\ (b\#bs) =$
$\quad (if\ a = b\ then\ None\ \#\ hideDupsRec\ b\ bs$
$\quad else\ Some\ b\ \#\ hideDupsRec\ b\ bs)$

$hideDups :: 'a\ list \Rightarrow 'a\ option\ list$
$hideDups\ [] = []$
$hideDups\ (b\#bs) = Some\ b\ \#\ hideDupsRec\ b\ bs$

$indexToVertexList :: face \Rightarrow vertex \Rightarrow nat\ list \Rightarrow vertex\ option\ list$
$indexToVertexList\ f\ v\ is \equiv (hideDups\ [f^{k}\bullet v.\ k \in is])$

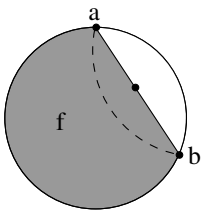
The function *indexToVertexList* is only applied on increasing index lists. Consider a vertex list *vs* obtained by converting an increasing index list to a vertex list using the function *indexToVertexList*. After removing all elements *None*, using the function *removeNones*, the vertices occur in the same order as in f starting at v. This is a precondition for the application of the function *addFace* to this vertex list.

Lemma *distinct* (*vertices* f) $\Longrightarrow$ $v \in$ *set* (*vertices* f) $\Longrightarrow$
incrIndexList es |*es*| |*vertices* f| $\Longrightarrow$
[$x \in$ *verticesFrom* f v. $x \in$ *set* (*removeNones* (*indexToVertexList* f v *es*))]
= *removeNones* (*indexToVertexList* f v *es*) $\surd$

Applying the Patches

Now we calculate the list of successor graphs from the enumerated patches. The function *generatePolygon* n v f g enumerates all index lists of length n for patches of a nonfinal face f of g, at a vertex v of f. Now, every index list is converted to a vertex list and the function *addFace* is called on every vertex list.

Before applying a patch for a nonfinal face f, it must be checked that none of the newly created edges (a, b) are already present in the graph, as we do not allow multiple edges. Multiple edges would violate the property of plane graphs that every edge occurs in a plane graph in exactly two faces with opposite directions (see Section 2.3).



The function *containsDuplicateEdge* tests if one of the edges of g would be duplicated if we applied an *addFace* operation to a vertex list obtained from an index list *is*. This situation can only occur if an edge (a, b) is introduced which joins two vertices already present in the face f, i.e. (a, b) does not contain new vertices. This is only the case if there is an index i in the index list *is* such that the predecessor p and the successor n are different from i. Then the edge joining $a = f^i \cdot v$ and $b = f^n \cdot v$ is introduced by an *addFace* operation (this is a property of the function *indexToVertexList*).

The condition *duplicateEdge* for a duplicate edge (a, b) states that it is already an edge of g, hence $b \in set\ (neighbors\ g\ a)$, and (a, b) is not an edge of f. Hence the distance between a and b is at least 2. Note that the case that (a, b) is an edge in f must not be excluded, because then *addFace* leaves the graph unchanged.

```
duplicateEdge :: graph ⇒ face ⇒ vertex ⇒ vertex ⇒ bool
duplicateEdge g f a b ≡
    let ab = directedLength f a b;
    ba = directedLength f b a in
    2 ≤ ba ∧ 2 ≤ ab ∧ b ∈ set (neighbors g a)

containsUnacceptableEdgeSnd :: (nat ⇒ nat ⇒ bool) ⇒ nat ⇒ nat list ⇒ bool
containsUnacceptableEdgeSnd N v [] = False
containsUnacceptableEdgeSnd N v (w#ws) =
    (case ws of [] ⇒ False
        | (w'#ws') ⇒ if v < w ∧ w < w' ∧ N w w' then True
            else containsUnacceptableEdgeSnd N w ws)

containsUnacceptableEdge :: (nat ⇒ nat ⇒ bool) ⇒ nat list ⇒ bool
containsUnacceptableEdge N [] = False
containsUnacceptableEdge N (v#vs) =
    (case vs of [] ⇒ False
        | (w#ws) ⇒ if v < w ∧ N v w then True
            else containsUnacceptableEdgeSnd N v vs)

containsDuplicateEdge :: graph ⇒ face ⇒ vertex ⇒ nat list ⇒ bool
containsDuplicateEdge g f v is ≡
    containsUnacceptableEdge (λi j. duplicateEdge g f (f^i• v) (f^j• v)) is
```

Finally, we define a function *generatePolygon* which calculates all successor graphs for nonfinal graphs by enumerating all possible patches and applying them to the graph:

```
generatePolygon :: 'parameter ⇒ nat ⇒ vertex ⇒ face ⇒ graph ⇒ graph list
generatePolygon_param n v f g ≡
    let enumeration = enumerator n |vertices f|;
    enumeration = [is ∈ enumeration. ¬ containsDuplicateEdge g f v is];
    vertexLists = [indexToVertexList f v is. is ∈ enumeration] in
    [addFace g f vs. vs ∈ vertexLists]
```

3.2.6 Inductive Definition of Plane Graphs

Now we define the set of plane graphs. Having defined a function which calculates all successor graphs for a nonfinal graph, we now define the tree of all graphs that are reachable by applying the successor function, starting with an initial graph.

We first define inductively a generic function *tree* as the reachability relation induced by a given successor function *succs*. An intuitive definition of the set of reachable graphs from a start graph g is the following: g is reachable from g (rule *root*). If g' is reachable from g and g'' is one of the successors of g' then g'' is reachable from g (rule *succs*).

We define a function $Tree :: (graph \Rightarrow graph\ list) \Rightarrow graph \Rightarrow graph\ set$ by the following two clauses:

root: $g \in Tree\ succs\ g$
succs: $g' \in Tree\ succs\ g \Longrightarrow g'' \in set\ (succs\ g') \Longrightarrow g'' \in Tree\ succs\ g$

We aim at a definition of plane graphs from which we can generate executable ML code. However, the ML code generated from this first definition of *Tree* does not terminate, even if the defined set is finite. This is due to the depth-first evaluation strategy used for inductive definitions [6]. When all elements of a finite set are enumerated, the generated function can still recursively call itself, such that the termination condition is never reached.

For this reason, we need to change the order of the premises in the induction step. Now, the evaluation is stopped as soon as $set\ (succs\ g)$ is empty. This does not allow us to treat the initial graph as a constant in the definition, hence we need to define trees as a binary relation of graphs rather than a function from graphs to a list of graphs. We end up with the following definition (see Figure 3.4):

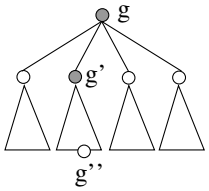


Figure 3.4: Inductive definition of graph trees.

We define a function *tree* by the following two clauses:

$tree :: (graph \Rightarrow graph\ list) \Rightarrow (graph \times graph)\ set$
$(g, g) \in tree\ succs$
$g' \in set\ (succs\ g) \Longrightarrow (g', g'') \in tree\ succs \Longrightarrow (g, g'') \in tree\ succs$

Now we define the set of all terminal (final) graphs in a tree, generated by a given *parameter param*, and with given *seed* and *succs* functions.

$terminalsTreeParam :: 'parameter \Rightarrow ('parameter \Rightarrow graph) \Rightarrow$
$('parameter \Rightarrow graph \Rightarrow graph\ list) \Rightarrow graph\ set$
$(seed\ param, g) \in tree\ (succs\ param) \Longrightarrow final\ g \Longrightarrow$
$g \in terminalsTree_{param}\ seed\ succs$

Then the set of all terminal graphs is the set of all graphs generated by any parameter:

$terminalsTree ::$
$('parameter \Rightarrow graph) \Rightarrow ('parameter \Rightarrow graph \Rightarrow graph\ list) \Rightarrow graph\ set$
$param: g \in terminalsTree_{param}\ seed\ succs \Longrightarrow$
$g \in terminalsTree\ seed\ succs$

We show that both definitions *tree* and *Tree* are equivalent, hence we can use the induction principles of both definition. For the proof see Appendix B.

Lemma *tree-eq*: $((g, g') \in tree\ succs) = (g' \in Tree\ succs\ g)$ $\surd$

For the definition of plane graphs it is sufficient to start generation with a seed graph consisting of a single face of arbitrary length and restrict the length of new faces to the length of the initial face.

Every seed graph is represented by a parameter which is the length of the final face, i.e. a natural number not smaller than 3:

$planeparameter = \{i::nat.\ 3 \leq i\}$

The function $Seed_{param}$ constructs a seed graph for the parameter *param*. The successor function $successorsList_{param}\ g$ calculates all patches for a graph g for all nonfinal faces f in g at all edges given by a vertex v in f, with a new final face of length i between 3 and the maximum face length given by the parameter *param*. Note that the functions *Seed* and *maxGon* are overloaded, i.e. for every parameter type there is a specific definition, whereas the function *successorsList* is a generic function and can be used for any parameter type. Using these functions, we finally define plane graphs.

$planeSeed\text{-}def\colon Seed_{m::planeparameter} \equiv graph\ (toNat\ m)$
$planeMaxGon\text{-}def\colon maxGon_{m::planeparameter} \equiv toNat\ m$

$successorsList :: 'parameter \Rightarrow graph \Rightarrow graph\ list$
$successorsList_{param}\ g \equiv if\ final\ g\ then\ []$
$\quad else\ let\ polylimit = maxGon_{param}\ in$
$\quad \cup_{f \in nonFinals\ g} \cup_{v \in vertices\ f} \cup_{i \in [3..polylimit]}\ generatePolygon_{param}\ i\ v\ f\ g$

$PlaneGraphs :: graph\ set$
$PlaneGraphs \equiv terminalsTree\ (Seed::planeparameter \Rightarrow graph)\ successorsList$

$PlaneGraphsParam :: planeparameter \Rightarrow graph\ set$
$PlaneGraphs_{param} \equiv$
$\quad terminalsTree_{param}\ (Seed::planeparameter \Rightarrow graph)\ successorsList$

$PlaneGraphsTree :: planeparameter \Rightarrow graph\ set$
$PlaneGraphsTree_{param} \equiv Tree\ successorsList_{param}\ Seed_{param}$

This definition of plane graphs contains infinitely many trees and each tree has infinitely many elements. From the inductive definition of trees, executable ML code can be generated that traverses the tree by a depth-first-strategy [6]. Note that this algorithm is neither terminating nor suitable to enumerate the complete set of plane graphs, as the depth of the trees which represents all plane graphs is unbounded.

3.2.7 Isomorphism of Plane Graphs

This section is devoted to the formal definition of plane graph isomorphism. We introduce plane graph isomorphism based on the equivalence relation on faces. Then we simplify the graph representation for final plane graphs by eliminating redundant information and abstracting from the type of the faces. Finally, we present an executable isomorphism test and its correctness theorem.

Isomorphism of Graphs

A *proper isomorphism* of two plane graphs g_1 and g_2 is a bijection φ of the set vertices of g_1 into the set of vertices of g_2 that induces a bijection of the set of faces of g_1 into the set of faces of g_2. It suffices that φ is injective on the set of vertices of g_1 and the homomorphism on the set of faces induced

by φ is surjective, i.e. *set* (*faces* g_2) is the image of *set* (*faces* g_1) under φ modulo equivalence on faces.

The function *liftFace* φ is the morphism on faces induced by a morphism φ on vertices.

liftFace :: (*vertex* ⇒ *vertex*) ⇒ *face* ⇒ *face*
liftFace φ (*Face vs t*) = *Face* [φ *v*. *v* ∈ *vs*] *t*

isHom :: (*vertex* ⇒ *vertex*) ⇒ *face list* ⇒ *face list* ⇒ *bool*
isHom φ fs_1 fs_2 ≡ *set* ([*liftFace* φ *f*. *f* ∈ fs_1])//≅ = *set* fs_2 //≅

isPrIso :: (*vertex* ⇒ *vertex*) ⇒ *face list* ⇒ *face list* ⇒ *bool*
isPrIso φ fs_1 fs_2 ≡ *isHom* φ fs_1 fs_2 ∧ *inj-on* φ ($\bigcup f \in set\ fs_1$. *set* (*vertices f*))

Two plane graphs are *properly isomorphic* (written as $g_1 \simeq g_2$) if there exists a graph isomorphism of their face lists.

$fs_1 \simeq fs_2 \equiv \exists \varphi$. *isPrIso* φ fs_1 fs_2

For each plane graph we obtain the *opposite* (mirrored) plane graph by reversing the cyclic order in each face.

(*fs*::*face list*)^{op} ≡ [f^{op}. *f* ∈ *fs*]
(*Graph fs n f h b*)^{op} = (*Graph* (fs^{op}) *n* [fi^{op}. *fi* ∈ *f*] *h b*)

Two plane graphs g_1 and g_2 with face lists fs_1 and fs_2 are isomorphic (written as $g_1 \cong g_2$) if fs_1 is properly isomorphic to fs_2 or $fsg_2{}^{op}$.

$fs_1 \cong (fs_2 :: face\ list) \equiv fs_1 \simeq fs_2 \vee fs_1 \simeq (fs_2{}^{op})$
$g_1 \cong (g_2{::}graph) \equiv (faces\ g_1) \cong (faces\ g_2)$

Abstraction of Plane Graphs

In order to make the construction of partial plane graphs efficient, the data structure of partial plane graphs stores more information than needed. Once a final plane graph is generated, this information is not needed any more. Moreover, we can abstract from the face type (*Final*/*Nonfinal*), and use a simpler representation by lists of faces, where a face is simply represented by a list of vertices.

'*a fsgraph* = '*a list list*
fsgraph :: *graph* ⇒ *nat fsgraph*
fsgraph g ≡ [*vertices f*. *f* ∈ (*faces g*)]

An Executable Isomorphism Test for Plane Graphs

The following executable isomorphism test for plane graphs was developed by Tobias Nipkow [5]. Morphisms are represented as a list of pairs. The function *test* checks if two morphisms are compatible and the function *merge* merges two compatible morphisms. The isomorphism test tries to pair faces of the same length and iterates over all rotations of the two faces. If the current isomorphism can be extended by some morphism I' (the result of a new pairing of two faces) then the function continues, otherwise it fails.

$$
\begin{array}{l}
\mathit{test} :: ('a, 'b)\ \mathit{table} \Rightarrow ('a, 'b)\ \mathit{table} \Rightarrow \mathit{bool} \\
\mathit{test}\ I\ I' \equiv \forall\, xy \in \mathit{set}\ I.\ \forall\, xy' \in \mathit{set}\ I'.\ (\mathit{fst}\ xy = \mathit{fst}\ xy') = (\mathit{snd}\ xy = \mathit{snd}\ xy')
\end{array}
$$

$$
\begin{array}{l}
\mathit{merge} :: ('a, 'b)\ \mathit{table} \Rightarrow ('a, 'b)\ \mathit{table} \Rightarrow ('a, 'b)\ \mathit{table} \\
\mathit{merge}\ I'\ I\ \equiv\ [xy \in I'.\ \forall\, xy' \in \mathit{set}\ I.\ \mathit{fst}\ xy \neq \mathit{fst}\ xy']\ @\ I
\end{array}
$$

$$
\begin{array}{l}
\mathit{pr\text{-}iso\text{-}test} :: ('a, 'b)\ \mathit{table} \Rightarrow 'a\ \mathit{fsgraph} \Rightarrow 'b\ \mathit{fsgraph} \Rightarrow \mathit{bool} \\
\mathit{pr\text{-}iso\text{-}test}\ I\ []\ \mathit{Fs}_2 = (\mathit{Fs}_2 = []) \\
\quad \mathit{pr\text{-}iso\text{-}test}\ I\ (F_1 \# \mathit{Fs}_1)\ \mathit{Fs}_2 = \\
\quad (\exists\, F_2 \in \mathit{set}\ \mathit{Fs}_2.\ |F_1| = |F_2| \wedge \\
\quad (\exists\, n{<}|F_2|. \\
\quad \mathit{let}\ I' = \mathit{zip}\ F_1\ (\mathit{rotate}\ n\ F_2)\ \mathit{in} \\
\quad \mathit{test}\ I'\ I \wedge \mathit{pr\text{-}iso\text{-}test}\ (\mathit{merge}\ I'\ I)\ \mathit{Fs}_1\ (\mathit{remove1}\ F_2\ \mathit{Fs}_2)))
\end{array}
$$

To improve the performance of the generated ML code, we first test if the two graphs have the same number of vertices and faces. To allow a quick check, we extend the graph representation by the number of vertices, which we can obtain from the representation of partial plane graphs.

$$
\begin{array}{l}
\mathit{pr\text{-}iso\text{-}test'} :: (\mathit{nat} \times 'a\ \mathit{fsgraph}) \Rightarrow (\mathit{nat} \times 'b\ \mathit{fsgraph}) \Rightarrow \mathit{bool} \\
\mathit{pr\text{-}iso\text{-}test'} \equiv \lambda(n_1,\mathit{Fs}_1)\ (n_2,\mathit{Fs}_2).\ n_1 = n_2 \wedge |\mathit{Fs}_1| = |\mathit{Fs}_2| \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{pr\text{-}iso\text{-}test}\ []\ \mathit{Fs}_1\ \mathit{Fs}_2
\end{array}
$$

We obtain the following correctness theorem for the proper isomorphism test on abstracted graphs. The isomorphism test works only under the assumption that for the representation of graphs certain conditions hold:

- The list of faces does not contain duplicate faces (modulo equivalence).
- Every face does not contain duplicate vertices.
- The number of vertices in a graph is calculated correctly.

Lemma $\forall F \in set\ Fs_1.\ distinct\ F \Longrightarrow$
$\forall F \in set\ Fs_2.\ distinct\ F \Longrightarrow [] \notin set\ Fs_2 \Longrightarrow$
$distinct\ Fs_1 \Longrightarrow inj\text{-}on\ (\lambda xs.\{xs\}//EqF)\ (set\ Fs_1) \Longrightarrow$
$distinct\ Fs_2 \Longrightarrow inj\text{-}on\ (\lambda xs.\{xs\}//EqF)\ (set\ Fs_2) \Longrightarrow$
$n_1 = card(\bigcup_{F \in set\ Fs_1} set\ F) \Longrightarrow$
$n_2 = card(\bigcup_{F \in set\ Fs_2} set\ F) \Longrightarrow$
$(Fs_1 \simeq Fs_2) = pr\text{-}iso\text{-}test'\ (n_1, Fs_1)\ (n_2, Fs_2)$ $\surd$

All these assumptions hold for all generated plane graphs, shown by induction on the construction (see Section 3.3), and can be used to simplify the correctness theorem.

Lemma $g_1 \in PlaneGraphsTree_{param} \Longrightarrow g_2 \in PlaneGraphsTree_{param} \Longrightarrow$
$g_1 \simeq g_2 =$
$(let\ Fs_1 = (\ |vertices\ g_1|, fsgraph\ g_1);$
$Fs_2 = (\ |vertices\ g_2|, fsgraph\ g_2)\ in$
$pr\text{-}iso\text{-}test'\ Fs_1\ Fs_2)$ *!*

The archive provided by Thomas Hales contained 5486 graphs, among those also some isomorphic copies. This is harmless for the proof of completeness of the enumeration algorithm, but it means extra work for further verification steps of the Kepler conjecture. With the verified executable isomorphism test we generate a reduced archive from the archive of tame graphs provided by Hales. This reduced the archive to 3050 graphs.

We use the isomorphism test for the second test function to decide if for every graph generated by the Isabelle function *Enumeration* there is an isomorphic graph contained in the reduced archive.

Finally we introduce some notation, used for both faces and graphs.

$g \in\cong G \equiv (\exists h \in G.\ g \cong h)$
$g \notin\cong G \equiv \neg\ g \in\cong G$
$G \subseteq\cong H \equiv \forall g \in G.\ g \in\cong H$
$G =\cong H \equiv G \subseteq\cong H \wedge H \subseteq\cong G$
$G \subset\cong H \equiv G \subseteq\cong H \wedge (\exists g \in H.\ g \notin\cong G)$

3.3 Invariants

Invariants are properties that hold for all partial graphs during the construction of a plane graph. They are proved by induction on the construction. The are two different kinds of invariants.

- We need consistence properties of plane graphs that characterize well-formed graphs, since the data structure used for the representation of graphs is highly redundant.

- Moreover, by the inductive definition, planarity imposes more restrictions on plane graphs. Some of them can be expressed by local properties of plane graphs.

In every construction step, at least one final face is added to a partial plane graph. The elementary operation is *addFace*. To verify that a property holds for all graphs during the construction, it is sufficient to show that the property is preserved by *addFace*. We use the following induction principle.

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow$
$P\ Seed_{param} \Longrightarrow$
$(\bigwedge g\ f\ v\ i\ e\ is\ g'.\ f \in set\ (nonFinals\ g) \Longrightarrow 2 < i \Longrightarrow$
$v \in set\ (vertices\ f) \Longrightarrow$
$g' = addFace\ g\ f\ is \Longrightarrow$
$e \in set\ (enumerator\ i\ |vertices\ f|\ ) \Longrightarrow$
$is = indexToVertexList\ f\ v\ e \Longrightarrow P\ g') \Longrightarrow P\ g$ $\surd$

An *addFace* operation calls a sequence of face split operations *FaceDivisionGraph* and *makeFaceFinal* operations to make a face final. Hence, we may prove a property P if we can show that P is preserved under *makeFaceFinal* and *FaceDivisionGraph* operations, provided the preconditions of these operations hold. This is reflected in the following induction principle.

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow$
$P\ Seed_{param} \Longrightarrow$
$(\bigwedge g\ f.\ f \in set\ (faces\ g) \Longrightarrow P\ g \Longrightarrow P\ (makeFaceFinal\ f\ g)) \Longrightarrow$
$(\bigwedge g\ f\ ram1\ ram2\ n.\ newVs = [countVertices\ g..<countVertices\ g + n] \Longrightarrow$
$pre\text{-}FaceDivisionGraph\ g\ ram1\ ram2\ f\ newVs \Longrightarrow P\ g \Longrightarrow$
$(f_1, f_2, g') = FaceDivisionGraph\ g\ ram1\ ram2\ f\ (rev\ newVs) \Longrightarrow$
$P\ g') \Longrightarrow$
$P\ g$ $\surd$

Using this induction principle simplifies a proof by induction, as it is not necessary to reason about the more complex *addFace* operation. However, not all properties may be proved using this induction principle, e.g. the property of a plane graph to contain no two adjacent nonfinal faces.

Consistence properties

The following properties are consistence properties of the data structure. The first follows from the definition of *facesAt*; for the other properties it has to be be verified that they are preserved by the *splitFace* operation.

Every face in the incidence list of a vertex v contains the vertex v.

Lemma $f \in set\ (facesAt\ g\ v) \Longrightarrow v \in set\ (vertices\ g)$ $\surd$

Every vertex v contained in a face f is also an element of the graph vertex list.

Lemma $g \in PlaneGraphs \Longrightarrow f \in set\ (faces\ g) \Longrightarrow v \in set\ (vertices\ f) \Longrightarrow$
$v \in set\ (vertices\ g)$ $\surd$

The list of incident faces with a vertex v consists of exactly the faces that contain v.

Lemma $g \in PlaneGraphs \Longrightarrow v \in set\ (vertices\ g) \Longrightarrow$
$set\ (facesAt\ g\ v) = \{f.\ f \in set\ (faces\ g) \wedge v \in set\ (vertices\ f)\}$ $\surd$

Finality

All terminal graphs in a tree of an arbitrary successor function are final by construction. The first property follows from the definition, the others follow from the definition of terminal graphs.

Lemma $finalVertex\ g\ v \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow final\ f$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow final\ g$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow v \in set\ (vertices\ g) \Longrightarrow finalVertex\ g\ v$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow final\ f$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow f \in set\ (faces\ g) \Longrightarrow final\ f$ $\surd$

Distinctness of Vertices and Faces

Preconditions for correctness of most of the functions used for the definition of the graph modification operations like *splitFace* and *addFace* are distinctness and non-emptiness of the list of vertices or faces. Moreover, every face must contain at least 3 vertices. The face list must not contain two equivalent faces, i. e. the list of normed faces must be distinct.

Again, it must be shown that these invariants are preserved by the *splitFace* operation.

Lemma $g \in PlaneGraphs \Longrightarrow set\ (vertices\ g) \neq \{\}$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow set\ (faces\ g) \neq \{\}$ *!*

Lemma $g \in PlaneGraphs \Longrightarrow v \in set\ (vertices\ g) \Longrightarrow$
$set\ (facesAt\ g\ v) \neq \{\}$ *!*

Lemma $g \in PlaneGraphs \Longrightarrow f \in set\ (faces\ g) \Longrightarrow 3 \leq |\ vertices\ f\ |$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow distinct\ (vertices\ g)$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow distinct\ [normFace\ f.\ f \in faces\ g]$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow distinct\ (faces\ g)$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow distinct\ [normFace\ f.\ f \in facesAt\ g\ v]$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow distinct\ (facesAt\ g\ v)$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow f \in set\ (faces\ g) \Longrightarrow distinct\ (vertices\ f)$ $\surd$

Lemma $g \in PlaneGraphs \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow distinct\ (vertices\ f)$ $\surd$

The following properties hold for all partial graphs during the construction. Navigation operations in graphs are well-defined. The next vertex $f{\cdot}v$ and the previous vertex $f^{-1}{\cdot}v$ of a vertex v in a face f are again in f.

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow v \in set\ (vertices\ g) \Longrightarrow$
$f \in set\ (facesAt\ g\ v) \Longrightarrow f{\cdot}v \in set\ (vertices\ g)$ *!*

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow v \in set\ (vertices\ g) \Longrightarrow$
$f \in set\ (facesAt\ g\ v) \Longrightarrow f^{-1}{\cdot}v \in set\ (vertices\ g)$ *!*

The next face $(g,v)\cdot f$ and the previous vertex $(g,v)^{-1}\cdot f$ of a face f incident with v are again incident with v.

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow (g,v)\cdot f \in set\ (facesAt\ g\ v)$ *!*

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow (g,v)^{-1}\cdot f \in set\ (facesAt\ g\ v)$ *!*

The next face $(g,v)\cdot f$ of a face f incident with v is also incident with the next vertex $w = f\cdot v$ of v in f and f is incident with w.

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow f\cdot v = w \Longrightarrow (g,v)\cdot f \in set\ (facesAt\ g\ w)$ *!*

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow f\cdot v = w \Longrightarrow f \in set\ (facesAt\ g\ w)$ *!*

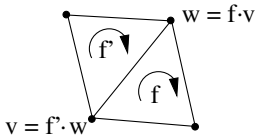
Plane graphs do not contain loops.

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (facesAt\ g\ a) \Longrightarrow f\cdot a \neq a$ *!*

If we repeatedly move to the next vertex in a face of length n, starting at v, after n iterations we return to v.

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (faces\ g) \Longrightarrow v \in set\ (vertices\ f) \Longrightarrow f^{length(vertices\ f)}\cdot v = v$ *!*

The following invariants are imposed by planarity of graphs.



Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow f\cdot v = w \Longrightarrow (g,v)\cdot f = f' \Longrightarrow f'\cdot w = v$ *!*

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f' \in set\ (facesAt\ g\ v) \Longrightarrow f = (g,v)^{-1}\cdot f' \Longrightarrow\ f\cdot v = f'^{-1}\cdot v$ *!*

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow f\cdot v = w \Longrightarrow (g,v)\cdot f = f' \Longrightarrow (g,\ w)^{-1}\cdot f = f'$ *!*

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow f \in set\ (facesAt\ g\ v) \Longrightarrow$
$(g,v)\cdot f = f' \Longrightarrow (v,\ w) \in set\ (edges\ \ f) \Longrightarrow (w,\ v) \in set\ (edges\ f')$ *!*

Lemma $g \in PlaneGraphsTree_{param} \Longrightarrow$
$f \in set\ (faces\ g) \Longrightarrow f' \in set\ (faces\ g) \Longrightarrow$
$(v,\ w) \in set\ (edges\ f) \Longrightarrow (v,\ w) \in set\ (edges\ f') \Longrightarrow f = f'$ *!*

A corollary is the property of plane graphs that every edge is contained in exactly two faces in opposite directions. Every edge (v, w) of a graph g is contained in one of the faces of g. Then (w, v) is only contained in $f' = (g,v)\cdot f$.

corollary $g \in PlaneGraphsTree_{param} \Longrightarrow (v,\ w) \in set\ (edges\ g) \Longrightarrow$
$(\exists!f.\ (v,\ w) \in set\ (edges\ f)) \wedge (\exists!f'.\ (w,\ v) \in set\ (edges\ f'))$ *!*

Euler's formula holds for all plane graphs, which can be proved by induction on the construction of a plane graph.

Theorem *Euler*:
$card\ (set(vertices\ g)) + card\ (set\ (faces\ g)) - card\ (set\ (edges\ g)) = 2$ *!*

Since the set of graphs to be considered in the proof of the Kepler conjecture is finite, these invariants can also be verified by a finite exhaustion. This can be achieved by introducing a test function in Isabelle which checks if the invariants hold for a given graph, and from which executable ML code can be generated. During generation, every generated partial graph can be tested for the invariants. This does not help for the proof of correctness of the enumeration though, since the properties are only verified for the enumerated graphs, not for all plane graphs. But the correctness proof relies on these properties for all plane graphs.

However, instead of actually verifying the invariants, this may be a useful test strategy for the graph operations. A good coverage with significant test cases may be expected since the selection of test cases is independent from the invariants. Whenever a graph is neglected, the test for the invariants is also carried out. A test strategy that generates arbitrary objects of graph type (as could be done for example with *quickcheck* [7], [**?**]) is not suitable to verify invariants, since the probability for an arbitrary graph object to hold the invariant is nearly 0. Hence almost no graph will be found for which the modification can be executed.

Chapter 4

Tame Plane Graphs

In this chapter we first recall the definition of tame plane graphs according to [24] (see Section 4.1) and then show how this definition of tameness can be defined in Isabelle/HOL (see Sections 4.2, 4.3, 4.4 and 4.5).

However, during the formalization it turned out that the definition must be modified: The original definition of tame graphs allowed graphs that were not generated by the enumeration program. Hence the enumeration program is not complete according to the original definition in [24]. Fortunately it can be shown that none of these graphs gives rise to a possible counterexample to the Kepler conjecture. Therefore the definition of tame plane graphs is changed in the way that these graphs are explicitely excluded (see Section 4.5).

4.1 Original Definition

In this section we present the definition of tame graphs according to [24]. First we need to define several constants and functions:

The constant 14.8 is called the *target*.

$\mathbf{a} : \mathbf{N} \to \mathbf{R}$ is defined by

$$\mathbf{a}(n) = \begin{cases} 14.8 & n = 0, 1, 2, \\ 1.4 & n = 3, \\ 1.5 & n = 4, \\ 0 & \text{otherwise} \end{cases}$$

$\mathbf{b} : \mathbf{N} \times \mathbf{N} \to \mathbf{R}$ is defined by the following table (where x=target=14.8), for all other values the result is 14.8.

$\mathbf{b}$(t,q)	$q = 0$	1	2	3	4	5
$t = 0$	x	x	x	7.135	10.649	x
1	x	x	6.95	7.135	x	x
2	x	8.5	4.756	12.981	x	x
3	x	3.642	8.334	x	x	x
4	4.139	3.781	x	x	x	x
5	0.55	11.22	x	x	x	x
6	6.339	x	x	x	x	x
7	x	x	x	x	x	x

$\mathbf{c} : \mathbf{N} \to \mathbf{R}$ is defined by

$$\mathbf{c}(n) = \begin{cases} 1 & n = 3, \\ 0 & n = 4, \\ -1.03 & n = 5, \\ -2.06 & n = 6, \\ -3.03 & \text{otherwise} \end{cases}$$

$\mathbf{d} : \mathbf{N} \to \mathbf{R}$ is defined by

$$\mathbf{d}(n) = \begin{cases} 0 & n = 3, \\ 2.378 & n = 4, \\ 4.896 & n = 5, \\ 7.414 & n = 6, \\ 9.932 & n = 7, \\ 10.916 & n = 8, \\ 14.8 & \text{otherwise} \end{cases}$$

A set of vertices is V called a *separated* set of vertices, iff

1. for every vertex in V there is an exceptional face containing it,

2. no two vertices in V are adjacent,

3. no two vertices in V lie on a common quadrilateral, and

4. each vertex in V has degree 5.

A *weight assignment* is a function $w : G \to \mathbf{R}_0^+$. A weight assignment is *admissible*, iff

1. $\mathbf{d}(|f|) \leq w(f)$,

2. if v has type (p, q), then $\mathbf{b}(p, q) \leq \sum_{v \in f} w(f)$,

3. let V be any set of vertices of type (5,0),
   if the cardinality k of V is ≤ 4, then $0.55k \leq \sum_{V \cap f \neq \emptyset} w(f)$,

4. let V be any separated set of vertices, and
   then $\sum_{v \in V} \mathbf{a}(tri(v)) \leq \sum_{V \cap f \neq \emptyset} (w(f) - d(|f|))$.

Definition. A plane graph is called *tame*, if it satisfies the following conditions:

1. The length of each face is at least 3 and at most 8.

2. Every 3-cycle is a face or the opposite of a face.

3. Every 4-cycle surrounds one of the cases illustrated in Figure 4.1.

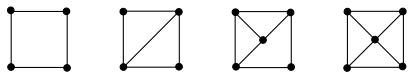


Figure 4.1: Tame 4-cycles

4. The degree of every vertex is at least 2 and at most 6.

5. If a vertex is contained in an exceptional face, then the degree of the vertex is at most 5.

6. The following inequality holds:

$$\sum_{f \in \mathcal{F}} \mathbf{c}(|f|) \geq 8.$$

7. There exists an admissible weight assignment of total weight $\sum_f w(f)$ less than the target, 14.8.

4.2 Constants

We show the representation of these definitions in Isabelle/HOL. The names of the defined constants correspond to the numbers of properties in the definition, e.g. $tame_3$ corresponds to property 3 of *tame*. The representation is quite close to the mathematical description for the conditions of tameness that can be expressed in set theoretic formulas. Other conditions, e.g. "all quadrilaterals surround a certain set of configurations", must be modeled explicitely. We multiply all constants by 1000 in order to calculate with integer values, since no higher precision ever occurs in the program.

```
squanderTarget :: nat
squanderTarget ≡ 14800

excessTCount :: nat ⇒ nat
a t ≡ if t < 3 then squanderTarget
    else if t = 3 then 1400
    else if t = 4 then 1500
    else 0

squanderVertex :: nat ⇒ nat ⇒ nat
b p q ≡ if p = 0 ∧ q = 3 then 7135
    else if p = 0 ∧ q = 4 then 10649
    else if p = 1 ∧ q = 2 then  6950
    else if p = 1 ∧ q = 3 then  7135
    else if p = 2 ∧ q = 1 then  8500
    else if p = 2 ∧ q = 2 then  4756
    else if p = 2 ∧ q = 3 then 12981
    else if p = 3 ∧ q = 1 then  3642
    else if p = 3 ∧ q = 2 then  8334
    else if p = 4 ∧ q = 0 then  4139
    else if p = 4 ∧ q = 1 then  3781
```

```
    else if p = 5 ∧ q = 0 then   550
    else if p = 5 ∧ q = 1 then 11220
    else if p = 6 ∧ q = 0 then  6339
    else squanderTarget

scoreFace :: nat ⇒ int
c n ≡ if n = 3 then 1000
    else if n = 4 then 0
    else if n = 5 then −1030
    else if n = 6 then −2060
    else if n = 7 then −3030
    else if n = 8 then −3030
    else −3030

squanderFace :: nat ⇒ nat
d n ≡ if n = 3 then 0
    else if n = 4 then 2378
    else if n = 5 then 4896
    else if n = 6 then 7414
    else if n = 7 then 9932
    else if n = 8 then 10916
    else squanderTarget
```

4.3 Separated Sets of Vertices

A set of vertices V is *separated*, iff the following conditions hold:

1. For each vertex in V there is an exceptional face containing it:

```
separated1 :: graph ⇒ vertex set ⇒ bool
separated1 g V ≡ ∀ v ∈ V. except g v ≠ 0
```

2. No two vertices in V are adjacent:

```
separated2 :: graph ⇒ vertex set ⇒ bool
separated2 g V ≡ ∀ v ∈ V. ∀ f ∈ set (facesAt g v). f·v ∉ V
```

3. No two vertices lie on a common quadrilateral:

```
separated3 :: graph ⇒ vertex set ⇒ bool
separated3 g V ≡
   ∀ v ∈ V. ∀ f ∈ set (facesAt g v). |vertices f|≤4 ⟶ set (vertices f)∩V = {v}
```

A set of vertices is called *preseparated*, iff no two vertices are adjacent or lie on a common quadrilateral:

$preSeparated :: graph \Rightarrow vertex\ set \Rightarrow bool$
$preSeparated\ g\ V \equiv separated_2\ g\ V \wedge separated_3\ g\ V$

4. Every vertex in V has degree 5:

$separated_4 :: graph \Rightarrow vertex\ set \Rightarrow bool$
$separated_4\ g\ V \equiv \forall\, v \in V.\ degree\ g\ v = 5$

$separated :: graph \Rightarrow vertex\ set \Rightarrow bool$
$separated\ g\ V \equiv$
$\quad separated_1\ g\ V \wedge separated_2\ g\ V \wedge separated_3\ g\ V \wedge separated_4\ g\ V$

4.4 Admissible Weight Assignments

A weight assignment $w :: face \Rightarrow nat$ assigns a natural number to every face.

We formalize the admissibility requirements as follows:

1. $\mathbf{d}(|f|) \leq w(f)$:

$admissible_1 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$
$admissible_1\ w\ g \equiv \forall f \in set\ (faces\ g).\ \mathbf{d}\ |vertices\ f| \leq w\ f$

2. If v has type (p, q), then $\mathbf{b}(p, q) \leq \sum_{v \in f} w(f)$:

$admissible_2 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$
$admissible_2\ w\ g \equiv$
$\quad \forall\, v \in set\ (vertices\ g).\ except\ g\ v = 0 \longrightarrow$
$\quad \mathbf{b}\ (tri\ g\ v)\ (quad\ g\ v) \leq \sum_{f \in facesAt\ g\ v} w\ f$

3. Let V be any set of vertices of type (5,0).
If the cardinality of V is $k \leq 4$, then $0.55k \leq \sum_{V \cap f \neq \emptyset} w(f)$:

$admissible_3 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$
$admissible_3\ w\ g \equiv$
$\quad \forall\, V.\ card\ V \leq 4 \longrightarrow$
$\quad V \subseteq \{v.\ v \in set\ (vertices\ g) \wedge tri\ g\ v = 5 \wedge quad\ g\ v = 0\} \longrightarrow$
$\quad \sum_{f \in [f \in faces\ g.\ V \cap set\ (vertices\ f) \neq \{\}]} w\ f \leq 550 * card\ V$

4. Let V be any separated set of vertices.
Then $\sum_{v \in V} \mathbf{a}(tri(v)) \leq \sum_{V \cap f \neq \emptyset} (w(f) - d(|f|))$:

$admissible_4 :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$
$admissible_4\ w\ g \equiv$
$\forall\, V.\ separated\ g\ (set\ V) \longrightarrow$
$set\ V \subseteq set\ (vertices\ g) \longrightarrow$
$(\sum_{v \in V} \mathbf{a}\ (tri\ g\ v))$
$+ (\sum_{f \in [f \in faces\ g.\ \exists\, v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} \mathbf{d}\ |vertices\ f|\ )$
$\leq \sum_{f \in [f \in faces\ g.\ \exists\, v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} w\ f$

Finally we define admissibility of weights functions.

$admissible :: (face \Rightarrow nat) \Rightarrow graph \Rightarrow bool$
$admissible\ w\ g \equiv$
$admissible_1\ w\ g \wedge admissible_2\ w\ g \wedge admissible_3\ w\ g \wedge admissible_4\ w\ g$

4.5 Tameness

In the algorithm of generating all tame plane graphs a graph is neglected if it contains two adjacent vertices of type $(4, 0)$ (see Figure 4.2). Therefore the

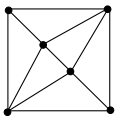
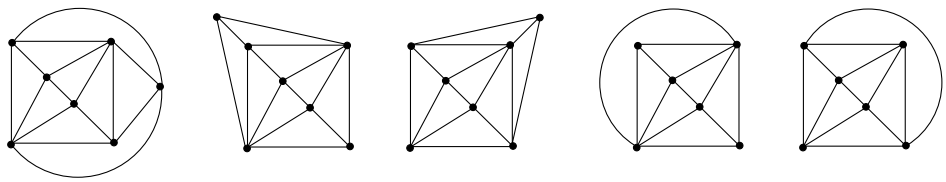


Figure 4.2: Two adjacent vertices of type $(4, 0)$

original definition of tame graphs as proposed Hales in fact includes graphs that are not generated by the algorithm. These graphs are of the following form: two adjacent vertices of type $(4, 0)$, bounded by a 4-cycle. On the outside one of the tame configurations of Figure 4.1, discarding any that give fewer than 8 triangles, for example the following graphs:



Hales [private communication] suggested to strengthen the notion of tameness in order to match the algorithm because it can be shown that all counterexamples must satisfy the stronger notion. Therefore we extended the definition of *tame* by a new restriction $tame_8$ that no two adjacent vertices of type $(4, 0)$ occur in a tame graph. Properties $tame_1$ to $tame_7$ correspond to properties 1 to 7 of the original definition.

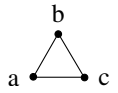
1. The length of each face is (at least 3 and) at most 8:

$$tame_1 :: graph \Rightarrow bool$$
$$tame_1\ g \equiv \forall f \in set\ (faces\ g).\ 3 \leq |vertices\ f| \wedge |vertices\ f| \leq 8$$

2. Every 3-cycle is a face or the opposite of a face:

A face given by a vertex list *vs* is contained in a graph g, if it is isomorphic to one of the faces in g. The notation $f \in_{\cong} F$ means $\exists f' \in F.\ f \cong f'$, where $\cong$ is the equivalence relation on faces (see Chapter 3.2.7).

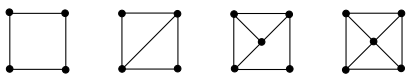
A *3-cycle* in a graph g is a cyclic path of length 3 along any faces of g such that the vertices of the path are distinct.



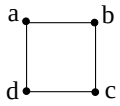
$$T3cycle :: vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow graph \Rightarrow bool$$
$$_3cycle\ a\ b\ c\ g \equiv distinct\ [a, b, c]$$
$$\wedge\ (\exists f \in set\ (faces\ g).\ (a, b) \in set\ (edges\ f))$$
$$\wedge\ (\exists f \in set\ (faces\ g).\ (b, c) \in set\ (edges\ f))$$
$$\wedge\ (\exists f \in set\ (faces\ g).\ (c, a) \in set\ (edges\ f))$$

$$tame_2 :: graph \Rightarrow bool$$
$$tame_2\ g \equiv$$
$$\forall a\ b\ c.\ _3cycle\ a\ b\ c\ g \longrightarrow$$
$$(Face\ [a, b, c]\ Final) \in_{\cong} set\ (faces\ g) \vee$$
$$(Face\ [c, b, a]\ Final) \in_{\cong} set\ (faces\ g)$$

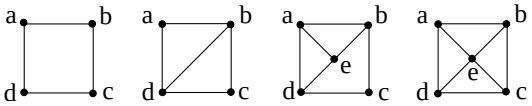
3. Every 4-cycle surrounds one of the following configurations:



A 4-cycle in a graph g is a cyclic path of length 4 along any faces of g, such that the vertices along the path are distinct.



$$
\begin{aligned}
&T4cycle :: vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow graph \Rightarrow bool\\
&_4cycle\ a\ b\ c\ d\ g \equiv distinct\ [a,\ b,\ c,\ d]\\
&\quad \wedge\ (\exists f \in set\ (faces\ g).\ (a,\ b) \in set\ (edges\ f))\\
&\quad \wedge\ (\exists f \in set\ (faces\ g).\ (b,\ c) \in set\ (edges\ f))\\
&\quad \wedge\ (\exists f \in set\ (faces\ g).\ (c,\ d) \in set\ (edges\ f))\\
&\quad \wedge\ (\exists f \in set\ (faces\ g).\ (d,\ a) \in set\ (edges\ f))
\end{aligned}
$$



$$
\begin{aligned}
&tameConf_1 :: vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow face\ set\\
&tameConf_1\ a\ b\ c\ d \equiv \{Face\ [a,\ b,\ c,\ d]\ Final\}
\end{aligned}
$$

$$
\begin{aligned}
&tameConf_2 :: vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow face\ set\\
&tameConf_2\ a\ b\ c\ d \equiv \{Face\ [a,\ b,\ c]\ Final,\ Face\ [a,\ c,\ d]\ Final\}
\end{aligned}
$$

$$
\begin{aligned}
&tameConf_3 :: vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow face\ set\\
&tameConf_3\ a\ b\ c\ d\ e \equiv\\
&\quad \{Face\ [a,\ b,\ e]\ Final,\ Face\ [b,\ c,\ e]\ Final,\ Face\ [a,\ e,\ c,\ d]\ Final\}
\end{aligned}
$$

$$
\begin{aligned}
&tameConf_4 :: vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow face\ set\\
&tameConf_4\ a\ b\ c\ d\ e \equiv\\
&\quad \{Face\ [a,\ b,\ e]\ Final,\ Face\ [b,\ c,\ e]\ Final,\ Face\ [c,\ d,\ e]\ Final,\\
&\quad Face\ [d,\ a\ ,\ e]\ Final\}
\end{aligned}
$$

Given a fixed 4-cycle and using the convention of drawing faces clockwise, a tame configuration can occur in the 'interior' or on the outside of the 4-cycle. For configuration $tameConf_2$ there are two possible rotations of the triangles, for configuration $tameConf_3$ there are 4. The notation $F_1 \subseteq\cong F_2$ means $\forall f \in F_1.\ f \in\cong F_2$.

Note that our definition only ensures the existence of certain faces in the graph, not the fact that no other faces of the graph may lie in the interior

or on the outside. Hence it is slightly weaker than the definition in Hales' paper.

$$
\begin{aligned}
&tame_4cycles :: graph \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow vertex \Rightarrow bool\\
&tame_4cycles\ g\ a\ b\ c\ d \equiv\\
&\quad \exists e.\ tameConf_1\ a\ b\ c\ d \subseteq\cong set\ (faces\ g)\\
&\qquad \vee\ tameConf_2\ a\ b\ c\ d \subseteq\cong set\ (faces\ g)\\
&\qquad \vee\ tameConf_2\ b\ c\ d\ a \subseteq\cong set\ (faces\ g)\\
&\qquad \vee\ tameConf_3\ a\ b\ c\ d\ e \subseteq\cong set\ (faces\ g)\\
&\qquad \vee\ tameConf_3\ b\ c\ d\ a\ e \subseteq\cong set\ (faces\ g)\\
&\qquad \vee\ tameConf_3\ c\ d\ a\ b\ e \subseteq\cong set\ (faces\ g)\\
&\qquad \vee\ tameConf_3\ d\ a\ b\ c\ e \subseteq\cong set\ (faces\ g)\\
&\qquad \vee\ tameConf_4\ a\ b\ c\ d\ e \subseteq\cong set\ (faces\ g)
\end{aligned}
$$

$$
\begin{aligned}
&tame_3 :: graph \Rightarrow bool\\
&tame_3\ g \equiv \forall a\ b\ c\ d.\ {}_4cycle\ a\ b\ c\ d\ g \longrightarrow\\
&\quad tame_4cycles\ g\ a\ b\ c\ d \vee tame_4cycles\ g\ d\ c\ b\ a
\end{aligned}
$$

4. The degree of every vertex is at least 2 and at most 6:

$$
\begin{aligned}
&tame_4 :: graph \Rightarrow bool\\
&tame_4\ g \equiv \forall v \in set\ (vertices\ g).\ 2 \le degree\ g\ v \wedge degree\ g\ v \le 6
\end{aligned}
$$

5. If a vertex is contained in an exceptional face, then the degree of the vertex is at most 5:

$$
\begin{aligned}
&tame_5 :: graph \Rightarrow bool\\
&tame_5\ g \equiv\\
&\quad \forall f \in set\ (faces\ g).\ \forall v \in set\ (vertices\ f).\ 5 \le |vertices\ f| \longrightarrow degree\ g\ v \le 5
\end{aligned}
$$

6. The following inequality holds:

$$
\begin{aligned}
&tame_6 :: graph \Rightarrow bool\\
&tame_6\ g \equiv 8000 \le \textstyle\sum_{f \in faces\ g} \mathbf{c}\ |vertices\ f|
\end{aligned}
$$

Note that this property implies that there are at least 8 triangles in a tame graph.

7. There exists an admissible weight assignment of total weight less than the target:

$$
\begin{aligned}
&tame_7 :: graph \Rightarrow bool\\
&tame_7\ g \equiv \exists w.\ admissible\ w\ g \wedge \textstyle\sum_{f \in faces\ g} w\ f < squanderTarget
\end{aligned}
$$

Property $tame_7$ implies that the set of tame plane graphs is finite.

8. We formalize the additional restriction (compared with the original definition) that tame graphs do not contain two adjacent vertices of type $(4, 0)$:

$type4\theta :: graph \Rightarrow vertex \Rightarrow bool$
$type4\theta\ g\ v \equiv$
$\quad tri\ g\ v = 4 \wedge quad\ g\ v = 0 \wedge except\ g\ v = 0$

$hasAdjacent4\theta :: graph \Rightarrow bool$
$hasAdjacent4\theta\ g \equiv$
$\quad \exists\, v \in set\ (vertices\ g).\ type4\theta\ g\ v \wedge$
$\quad (\exists\, w \in set\ (neighbors\ g\ v).\ type4\theta\ g\ w)$

$tame_8 :: graph \Rightarrow bool$
$tame_8\ g \equiv \neg\ hasAdjacent4\theta\ g$

Finally we define the notion of tameness.

$tame :: graph \Rightarrow bool$
$tame\ g \equiv$
$\quad tame_1\ g \wedge tame_2\ g \wedge tame_3\ g \wedge tame_4\ g \wedge tame_5\ g \wedge tame_6\ g \wedge tame_7\ g$
$\wedge\ tame_8\ g$

Chapter 5

Enumeration of Tame Plane Graphs

Starting from the definition of plane graphs we obtain a first algorithm to enumerate all tame plane graphs: enumerate all plane graphs (by breadth-first-strategy) and remove graphs that are definitely not tame. This first algorithm is not terminating, since the set of all plane graphs is not finite.

However, the definition of tameness implies that the set of tame graphs is finite. Hence we use the following approach: We start with the set of all plane graphs (Ref_0 = PlaneGraphs in Figure. 5.1). We gradually reduce the generated set of graphs, imposing the restrictions of tameness to it (Ref_1, ..., Ref_4), such that the generated set of graphs eventually becomes finite, is efficiently enumerable, and still contains all tame graphs (and maybe some graphs that are not tame) (Ref_5 = *Enumeration*).

This leads to a series of proof obligations, i.e. the completeness of each refinement step. All completeness proofs together yield the completeness theorem for *Enumeration*.

Note that it is not necessary to remove all graphs that are not tame. It is sufficient to generate a superset of the set of tame plane graphs.

Basically, there are two different reasons why a graph g can be neglected (types of refinements):

(I) if all final graphs generated by g are not tame or

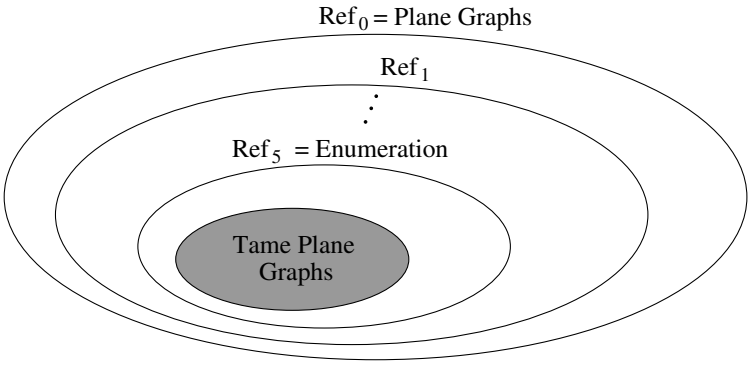


Figure 5.1: Proof structure

(II) if for every final graph generated by g an isomorphic graph will be generated by another path in the tree.

5.1 Fixing a Face and an Edge

In the definition of plane graphs we calculate successors for all edges in all nonfinal faces. As a first refinement step we fix one nonfinal face and one edge in this face and calculate the successors only for this edge. We denote the generated set of graphs by $PlaneGraphs_2$. This does not reduce the set of generated graphs modulo isomorphism. For all graphs which are left out, an isomorphic graph is generated by another path in the tree. This is an optimization of type (II). Hence this may still be considered as a definition for plane graphs. The completeness proof is by induction on the generation of a graph. It is sufficient to show that, modulo graph isomorphism, the generation does not depend on which nonfinal face f and which edge in f we apply the *addFace* operation first. The following functions select a certain nonfinal face and a certain vertex in a face.

The function *minimalTempFace g* returns a nonfinal face with minimal number of vertices, if a nonfinal face exists in g.

```
minimalTempFace:: graph ⇒ face option
minimalTempFace g ≡
    let fs = nonFinals g in
    if fs = [] then None
    else Some (minimal (length ∘ vertices) (hd fs) (fs))
```

An edge in a face f is uniquely determined by a vertex v, the endpoint of the edge. The function *minimalVertex g f* returns a vertex with minimal height in the face f of g. The height of a vertex is the distance to one of the vertices in the seed a graph was generated from. The selection of a vertex of minimal height has the side effect that the generated nonfinal graphs stay 'compact'.

```
minimalVertex :: graph ⇒ face ⇒ vertex
minimalVertex g f ≡ let vs = vertices f in minimal (height g) (hd vs) vs
```

The successor function of a nonfinal graph g selects a minimal face f in g and a vertex v of minimal height in f and generates the list of successor graphs, where a face of length i is created at vertex v in f, and i is between 3 and the maximal face length given by the parameter of the tree.

```
successorsList' :: 'parameter ⇒ graph ⇒ graph list
successorsList_param' g ≡
    if final g then []
    else let fopt = minimalTempFace g in
    case fopt of None ⇒ []
      | Some f ⇒
        let v = minimalVertex g f;
        polylimit = maxGon_param in
        ∪_{i∈[3..polylimit]} generatePolygon param i v f g
```

We define the alternative set of plane graphs:

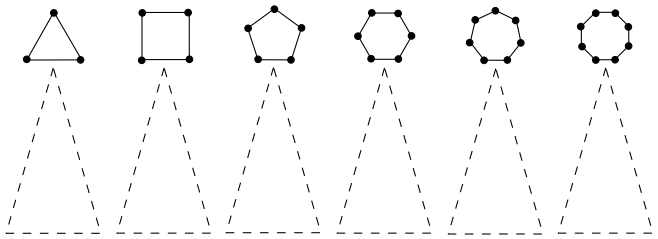
```
PlaneGraphsN :: graph set
PlaneGraphsN ≡ terminalsTree (Seed::planeparameter ⇒ graph) successorsList'

PlaneGraphsNParam :: planeparameter ⇒ graph set
PlaneGraphsN_param ≡
    terminalsTree_param (Seed::planeparameter ⇒ graph) successorsList'

PlaneGraphsNTree :: planeparameter ⇒ graph set
PlaneGraphsNTree_param ≡ Tree successorsList_param' Seed_param
```

For the completeness of this refinement we show that for every plane graph an isomorphic graph is generated by the optimized definition (see Section 6.2).

Theorem *planeN-complete*:
$g \in PlaneGraphs \Longrightarrow g \in_{\cong} PlaneGraphsN$

5.2 Restriction of Maximum Face Size to 8

Since all tame graphs have maximum face size 8 ($tame_1$, see Section 4.5), we can exclude all seed graphs with face size greater than 8. Every graph generated from these seeds will contain a face of size greater than 8 and hence will not be tame. This is a refinement of type (I).



We restrict the set of seed parameters to the finite set $\{3, \ldots, 8\}$.

$finparameter = \{3{::}nat..8\}$
$Seed_{m::finparameter} \equiv graph\ (toNat\ m)$
$maxGon_{m::finparameter} \equiv toNat\ m$

We define a restricted set of plane graphs with maximum face length 8.

$PlaneGraphs_2 :: graph\ set$
$PlaneGraphs_2 \equiv terminalsTree\ (Seed{::}finparameter \Rightarrow graph)\ successorsList$

$PlaneGraphs_2Param :: finparameter \Rightarrow graph\ set$
$PlaneGraphs_{2param} \equiv$
$\quad terminalsTree_{param}\ (Seed{::}finparameter \Rightarrow graph)\ successorsList$

$PlaneGraphs_2Tree :: finparameter \Rightarrow graph\ set$
$PlaneGraphs_2Tree_{param} \equiv Tree\ successorsList_{param}\ Seed_{param}$

For the correctness theorem, we need to show that we do not lose any tame graphs by this restriction (see Section 6.3).

Theorem *plane*$_2$*-complete*:
$g \in PlaneGraphsN \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} PlaneGraphs_2$

5.3 Complex Seed Graphs

In the next refinement step we replace the first two seed graphs (consisting of a final triangle and a final quadrilateral, respectively) by a new finite set

of complex seed graphs (see Figure 5.2). Every new seed graph has one final vertex v. It consists of one nonfinal face and a set of final triangles and final quadrilaterals, arranged around v.

This refinement is divided into the following steps: partitioning of quad parameters, introduction of complex seed graphs, restriction to tame seed graph, and neglecting graphs containing earlier seed graphs.

Partitioning of Quad Parameters

The first step is an equivalent definition to the previous one separating the set of parameters in two groups, *quad parameter* $\{3, 4\}$ and *exceptional parameter* $\{5, \ldots, 8\}$.

$qparameter = \{3::nat,4\}$
$exceptionalparameter = \{5::nat..8\}$

$qeparameter =$
  $Qparameter\ qparameter$
 $|\ Eparameter\ exceptionalparameter$

$Seed_{Qparameter\ (m::qparameter)} = graph\ (toNat\ m)$
$Seed_{Eparameter\ (m::exceptionalparameter)} = graph\ (toNat\ m)$

$maxGon_{Qparameter\ (m::qparameter)} = toNat\ m$
$maxGon_{Eparameter\ (m::exceptionalparameter)} = toNat\ m$

$PlaneGraphs_3 :: graph\ set$
$PlaneGraphs_3 \equiv terminalsTree\ (Seed::qeparameter \Rightarrow graph)\ successorsList$

$PlaneGraphs_3Param :: qeparameter \Rightarrow graph\ set$
$PlaneGraphs_{3param} \equiv\ terminalsTree_{param}\ Seed\ successorsList$

$PlaneGraphs_3Tree :: qeparameter \Rightarrow graph\ set$
$PlaneGraphs_3Tree_{param} \equiv Tree\ successorsList_{param}\ Seed_{param}$

For the proof of the completeness theorem see Section 6.4.

Theorem *plane*$_3$*-complete*:
$g \in$

Complex Seed Graphs

In the second step we replace the two quad seed graphs

by an (infinite) enumerable set of complex seed graphs (*vertex seed graphs*) with one nonfinal face and a set of final faces arranged around one final vertex v (of degree at least 2).

Each seed graph is represented by a *vertex parameter*, i.e. a list of quad parameters (of the length of the degree of v) representing the sizes of the faces incident with v in cyclic order. Hence we can enumerate all vertex seed graphs by constructing all graphs with one final vertex of degree n surrounded by all possible combinations of triangles and quadrilaterals, for $n = 2, 3, 4, \ldots$.

$vertexparameter = \{ns::qparameter\ list.\ 2 \leq length\ ns\}$
$veparameter =$
$\quad QParameter\ vertexparameter$
$\mid EParameter\ exceptionalparameter$

The function *SeedGraph* constructs a vertex seed graph from a list of lengths of faces.

$SeedGraph :: nat\ list \Rightarrow graph$
$Seed\ (QParameter\ (m::vertexparameter)) = SeedGraph\ (toNatList\ m)$
$Seed\ (EParameter\ (m::exceptionalparameter)) = graph\ (toNat\ m)$

The successors function is adjusted to add triangles as well and quadrilaterals: the maximal face length *maxGon* of faces that can be added is 4.

$maxGon_{QParameter\ (m::vertexparameter)} = 4$
$maxGon_{EParameter\ (m::exceptionalparameter)} = toNat\ m$

We define the set of plane graphs, generated by complex seed graphs.

$PlaneGraphs_4 :: graph\ set$
$PlaneGraphs_4 \equiv terminalsTree\ (Seed::veparameter \Rightarrow graph)\ successorsList$

$PlaneGraphs_4Param :: veparameter \Rightarrow graph\ set$
$PlaneGraphs_{4param} \equiv terminalsTree_{param}\ Seed\ successorsList$

$PlaneGraphs_4 Tree :: veparameter \Rightarrow graph\ set$
$PlaneGraphs_4 Tree_{param} \equiv Tree\ (successorsList_{param})\ (Seed_{param})$

Again, in this steps no graphs are excluded up to isomorphism and the correctness theorem is:

Theorem *plane*$_4$*-complete*:
$g \in PlaneGraphs_3 \Longrightarrow g \in_{\cong} PlaneGraphs_4$

For the proof see Section 6.4.

Restriction to Tame Complex Seed Graphs

As the next step we exclude seed graphs which will never lead to tame graphs using properties (*tame*$_7$) and (*admissible*$_2$).

We call (t, q) the *type* of a seed graph s iff $(p,\ q)$ is the type of the final vertex v in s. From the set of vertex seed graphs we exclude every seed graph s of type $(p,\ q)$ if $14.8 \leq \mathbf{b}\ p\ q$. Every graph g generated by s contains a vertex v with $14.8 \leq \mathbf{b}\ p\ q$ and for every admissible weight assignment w, $\mathbf{b}\ p\ q \leq \sum_{f \in faces\ g} w\ f$ (*admissible*$_2$). Hence there is no admissible weight assignment with total weight less than the target 14.8 for g and g is not tame (*tame*$_7$).

Therefore we make a list of all types (p, q) with $14.8 \leq \mathbf{b}\ p\ q$ and create all seed graphs (up to isomorphism) of this type. This results in a list of 17 seed graphs shown in Figure 5.2. Moreover we impose a fixed order on these seed graphs by assigning each graph an index out of the set $\{0, \ldots, 16\}$. Every graph is represented by this index as parameter.

$quadparameter = \{0..16::nat\}$
$parameter =$
  $QuadParameter\ quadparameter$
| $ExceptionalParameter\ exceptionalparameter$

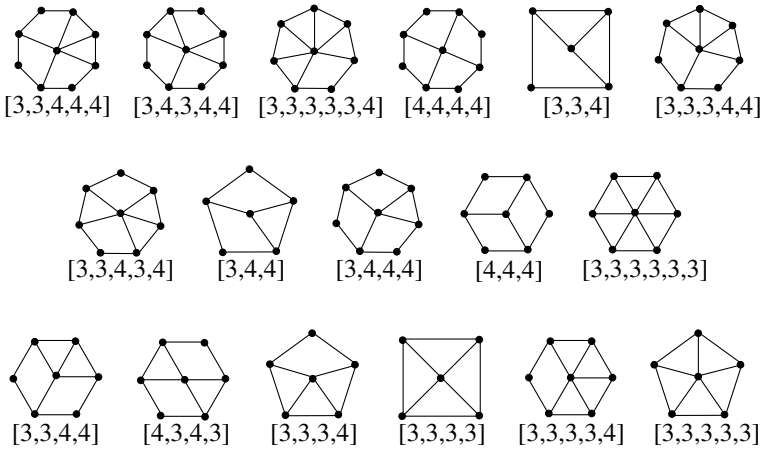


Figure 5.2: Complex Seed graphs with maximum face length of 4

$quadCase :: nat \Rightarrow nat\ list$
$quadCase\ i \equiv if\ i = 0\ then\ [3, 3, 4, 4, 4]$
$\quad else\ if\ i = 1\ then\ [3, 4, 3, 4, 4]$
$\quad else\ if\ i = 2\ then\ [3, 3, 3, 3, 3, 4]$
$\quad else\ if\ i = 3\ then\ [4, 4, 4, 4]$
$\quad else\ if\ i = 4\ then\ [3, 3, 4]$
$\quad else\ if\ i = 5\ then\ [3, 3, 3, 4, 4]$
$\quad else\ if\ i = 6\ then\ [3, 3, 4, 3, 4]$
$\quad else\ if\ i = 7\ then\ [3, 4, 4]$
$\quad else\ if\ i = 8\ then\ [3, 4, 4, 4]$
$\quad else\ if\ i = 9\ then\ [4, 4, 4]$
$\quad else\ if\ i = 10\ then\ [3, 3, 3, 3, 3, 3]$
$\quad else\ if\ i = 11\ then\ [3, 3, 4, 4]$
$\quad else\ if\ i = 12\ then\ [4, 3, 4, 3]$
$\quad else\ if\ i = 13\ then\ [3, 3, 3, 4]$
$\quad else\ if\ i = 14\ then\ [3, 3, 3, 3]$
$\quad else\ if\ i = 15\ then\ [3, 3, 3, 3, 4]$
$\quad else\ if\ i = 16\ then\ [3, 3, 3, 3, 3]$
$\quad else\ []$

0	(2,3)
1	(2,3)
2	(5,1)
3	(0,4)
4	(2,1)
5	(3,2)
6	(3,2)
7	(1,2)
8	(1,3)
9	(0,3)
10	(6,0)
11	(2,2)
12	(2,2)
13	(3,1)
14	(4,0)
15	(4,1)
16	(5,0)

$Seed_{QuadParameter\ (n::quadparameter)} = SeedGraph\ (quadCase\ (toNat\ n))$
$Seed_{ExceptionalParameter\ (m::exceptionalparameter)} = graph\ (toNat\ m)$

$maxGon_{QuadParameter\ (m::quadparameter)} = 4$
$maxGon_{ExceptionalParameter\ (m::exceptionalparameter)} = toNat\ m$

$succsPlane_5 :: parameter \Rightarrow graph \Rightarrow graph\ list$
$succsPlane_{5\ param} \equiv successorsList\ param$

$PlaneGraphs_5 :: graph\ set$
$PlaneGraphs_5 \equiv terminalsTree\ (Seed{::}parameter \Rightarrow graph)\ successorsList$

$PlaneGraphs_5Param :: parameter \Rightarrow graph\ set$
$PlaneGraphs_{5param} \equiv terminalsTree_{param}\ Seed\ successorsList$

$PlaneGraphs_5Tree :: parameter \Rightarrow graph\ set$
$PlaneGraphs_5Tree_{param} \equiv Tree\ successorsList_{param}\ Seed_{param}$

This leads to the following completeness theorem (see Section 6.4).

Theorem *plane*$_5$*-complete*:
$g \in PlaneGraphs_4 \Longrightarrow tame\ g \Longrightarrow \exists h \in PlaneGraphs_5.\ g \cong h$

Graphs Containing Earlier Seed Graphs

Using the order we introduced on the set of seed graphs, we neglect a (non-final) graph g, generated by a seed s, if we can conclude that an isomorphic graph is generated by some earlier seed graph s' (see Figure 5.3). It is sufficient to check if g contains a final vertex of the same type as an earlier seed graph and of different type as s (*matchEarlierSeed*). If the faces around v form one of the tame configurations (see Figure 5.2), then there is a seed s' such that s' is a subgraph modulo isomorphism of g and for every graph g' generated from g an isomorphic graph is generated starting with s'. Otherwise, every graph g' generated from g is not tame, hence g can be neglected. This refinements avoids generating isomorphic copies of tame plane graphs.

The algorithm which was used to build the archive included this optimization in the following way: a modified function **b'** is used instead of **b** such that partial graphs generated by a *QuadParameter* may be neglected if they contain a final vertex which is isomorphic to an earlier seed graph with a different type than the current seed graph.

$tCount :: nat \Rightarrow nat$
$tCount\ n \equiv |\ [x \in quadCase\ n.\ x{=}3]\ |$

$qCount :: nat \Rightarrow nat$
$qCount\ n \equiv |\ [x \in quadCase\ n.\ x{=}4]\ |$

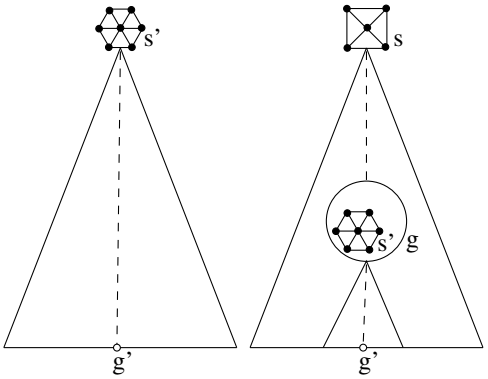


Figure 5.3: Neglect graphs that contain earlier seed graphs

$$
\begin{array}{l}
matchSeed :: nat \Rightarrow nat \Rightarrow nat \Rightarrow bool \\
matchSeed\ n\ t\ q \equiv (t = tCount\ n) \wedge (q = qCount\ n)
\end{array}
$$

$$
\begin{array}{l}
matchEarlierSeed :: nat \Rightarrow nat \Rightarrow nat \Rightarrow bool \\
matchEarlierSeed\ n\ t\ q \equiv \\
\quad \neg\ matchSeed\ n\ t\ q \wedge (\exists\, m \in set\ [0\ ..{<}\ n].\ matchSeed\ m\ t\ q)
\end{array}
$$

$$
\begin{array}{l}
squanderVertexParam :: parameter \Rightarrow nat \Rightarrow nat \Rightarrow nat \\
\mathbf{b'}_{QuadParameter\ n}\ t\ q = \\
\quad (if\ matchEarlierSeed\ (toNat\ n)\ t\ q\ then\ squanderTarget\ else\ \mathbf{b}\ t\ q) \\
\mathbf{b'}_{ExceptionalParameter\ m}\ t\ q = \mathbf{b}\ t\ q
\end{array}
$$

For the verification we introduce an additional refinement step which excludes all graphs that contain earlier seeds.

$$
\begin{array}{l}
containsEarlierSeed :: parameter \Rightarrow graph \Rightarrow bool \\
containsEarlierSeed_{QuadParameter\ (n::quadparameter)}\ g = \\
\quad (\exists\, v \in set\ (vertices\ g).\ finalVertex\ g\ v \\
\qquad \wedge\ matchEarlierSeed\ (toNat\ n)\ (tri\ g\ v)\ (quad\ g\ v)) \\
containsEarlierSeed_{ExceptionalParameter\ (m::exceptionalparameter)}\ g = False
\end{array}
$$

$$
\begin{array}{l}
succsPlane_6 :: parameter \Rightarrow graph \Rightarrow graph\ list \\
succsPlane_6\ param\ g \equiv \\
\quad [g' \in successorsList_{param}\ g.\ \neg\ containsEarlierSeed_{param}\ g']
\end{array}
$$

$$
\begin{array}{l}
PlaneGraphs_6 :: graph\ set \\
PlaneGraphs_6 \equiv terminalsTree\ (Seed::parameter \Rightarrow graph)\ succsPlane_6
\end{array}
$$

$PlaneGraphs_6Param :: parameter \Rightarrow graph\ set$
$PlaneGraphs_{6param} \equiv terminalsTree_{param}\ Seed\ succsPlane_6$

$PlaneGraphs_6Tree :: parameter \Rightarrow graph\ set$
$PlaneGraphs_6Tree_{param} \equiv Tree\ succsPlane_6\ param\ Seed_{param}$

The completeness proof of this refinement step is expressed by the following theorem and can be found in Section 6.4.

Theorem *plane*$_6$*-complete*:
$g \in PlaneGraphs_5 \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} PlaneGraphs_6$

5.4 Neglectable by Base Point Symmetry

We may neglect graphs for which an isomorphic graph has been generated by another path in the tree. For example, if a partial graph g contains more than one exceptional face of maximal size, one of these faces is the seed face, and an isomorphic graph is generated from any other face of maximal size. Hence we may exclude one of these cases.

By construction, when a graph contains exceptional faces, the base vertex lies in the seed face. We calculate a hash value for every vertex, such that the vertex with the greatest hash lies on a face of maximal length in the graph. This may either be the base vertex itself or another vertex on the same face as the base vertex or on some other face of maximal length. We neglect a graph if the base vertex is not the vertex with the greatest hash.

We calculate a hash value *adjacentLengths* $g\ v$ for a vertex v in a graph g by the decreasing sorted list of the lengths of the adjacent faces. A hash as is smaller than a hash bs if as is lexicographically smaller than bs (*listLess* as bs).

$listLess :: nat\ list \Rightarrow nat\ list \Rightarrow bool$
$listLess\ []\ bs = (bs \neq [])$
$listLess\ (a\#as)\ bs = (case\ bs\ of\ [] \Rightarrow False$
$\quad | (b\#bs') \Rightarrow a < b \vee a = b \wedge listLess\ as\ bs')$

$adjacentLengths :: graph \Rightarrow vertex \Rightarrow nat\ list$
$adjacentLengths\ g\ v \equiv rev\ (qsort\ (op \leq, ([\ |vertices\ f|.\ f \in facesAt\ g\ v])))$

$neglectableByBasePointSymmetry :: graph \Rightarrow bool$
$neglectableByBasePointSymmetry\ g \equiv$

```
  let bv = baseVertex g in
  case bv of None ⇒ False
    | (Some v) ⇒ (let lv = adjacentLengths g v in
        (∃ w ∈ set(vertices g). |nonFinalsAt g w| = 0
          ∧ listLess lv (adjacentLengths g w)))

succsPlane γ :: parameter ⇒ graph ⇒ graph list
succsPlane γ param g ≡
  [g' ∈ successorsList_param g. ¬ neglectableByBasePointSymmetry g'
    ∧ ¬ containsEarlierSeed_param g']

PlaneGraphs γ :: graph set
PlaneGraphs γ ≡ terminalsTree (Seed::parameter ⇒ graph) succsPlane γ

PlaneGraphs γParam :: parameter ⇒ graph set
PlaneGraphs γ_param ≡ terminalsTree_param Seed succsPlane γ

PlaneGraphs γTree :: parameter ⇒ graph set
PlaneGraphs γTree_param ≡ Tree succsPlane γ param Seed_param
```

For the proof of the correctness theorem we have to show: if a graph g is neglected because the base vertex is not the vertex with the greatest hash, then for every graph g' generated from g, an isomorphic graph h is generated by another seed graph such that in h the base vertex is the vertex with the greatest hash.

Theorem *plane γ-complete*:
$g \in PlaneGraphs_6 \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} PlaneGraphs\,\gamma$

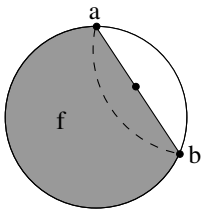
For the proof see Section 6.5.

5.5 Avoiding Vertices Enclosed by 3-Cycles

The function *neglectableEdge* determines whether the creation of an edge joining the vertices a and b in a nonfinal face f can be neglected. This is the case if

- the edge (a,b) is already present in the graph (*duplicateEdge*), i.e. a and b have distance at most 2 from each other and b is one of the neighbors of a.

This restriction is already introduced for plane graphs (see Section 3.2.5).



- the creation of an edge (a,b) in the graph g leads to a cycle of length 3, enclosing some vertices ($enclosedVertex$), which is not allowed by the property $tame_2$.

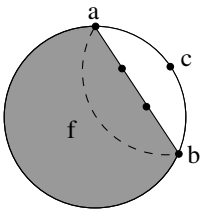
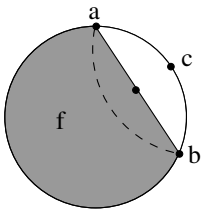


Figure 5.4: A 3-cycle with enclosed vertices.

```
duplicateEdge :: graph ⇒ face ⇒ vertex ⇒ vertex ⇒ bool
duplicateEdge g f a b ≡
    let ab = directedLength f a b;
    ba = directedLength f b a in
    (2 ≤ ba ∧ 2 ≤ ab) ∧ b ∈ set (neighbors g a)

enclosedVertex :: graph ⇒ face ⇒ vertex ⇒ vertex ⇒ bool
enclosedVertex g f a b ≡
    let ab = directedLength f a b;
    ba = directedLength f b a in
    3 ≤ ba ∧ 3 ≤ ab ∧ (neighbors g a ∩ neighbors g b) ≠ []

neglectableEdge :: graph ⇒ face ⇒ vertex ⇒ vertex ⇒ bool
neglectableEdge g f a b ≡ duplicateEdge g f a b ∨ enclosedVertex g f a b
```

Note that this function does not detect if the introduction of an edge (a, b) creates a cycle of length 3 and the distance of a and b on the face is less than 3. Hence this function could be further improved by testing if a and b have distance at least 2 from each other and they have at least 2 common neighbors.




```
enclosedVertex2 :: graph ⇒ vertex ⇒ vertex ⇒ face ⇒ bool
enclosedVertex2 g a b f ≡
    let ab = directedLength f a b;
    ba = directedLength f b a in
    3 ≤ ba ∧ 3 ≤ ab ∧ set (neighbors g a) ∩ set (neighbors g b) ≠ {}
    ∨  2 ≤ ba ∧ 2 ≤ ab ∧ 2 ≤ |neighbors g a ∩ neighbors g b|
```

Finally, we define a function *containsNeglectableEdge* which is used to determine for a list of indices *is* generated by the *enumerator* function if a neglectable edge would be introduced by an *addFace* operation on the corresponding vertex list.

Since the results of the function *neglectableEdge* are used several times, we store the results in a table *neglectableEdgeTable*.

neglectableEdgeTable To avoid duplicate calculations, we store the results of *neglectableEdge* in a table *neglectableEdgeTable* of index pairs. *neglectableEdgeTable*⟦*i*,*j*⟧ is *True* if an edge $(f^{i}\bullet v,\ f^{j}\bullet v)$ is neglectable, joining the vertices reachable from v in f by i and j steps.

```
neglectableEdgeTable :: graph ⇒ face ⇒ vertex ⇒ bool array array
neglectableEdgeTable g f v ≡
    ⟦ neglectableEdge g f (f^i•v) (f^j•v). i < |vertices f|, j < |vertices f| ⟧
```

containsNeglectableEdge For a given graph g, a nonfinal face f in g, and a vertex v in f, we calculate once the table of neglectable index pairs $T =$ *neglectableEdgeTable v f g*.

Given a modification of a graph g by an *addFace* operation in the nonfinal face f where the vertices of the new final face are given by an index list *is*, we determine if a neglectable edge would be introduced: we use the function *containsUnacceptableEdge* (see Section 3.2.5) and test if for all pairs $(i,\ j)$

of elements of *is* an *addFace* operation would introduce an edge joining two vertices in f, namely $f^{i}\!\cdot\! v$ and $f^{j}\!\cdot\! v$ and if this new edge is neglectable.

```
containsNeglectableEdge :: bool array array ⇒ nat list ⇒ bool
containsNeglectableEdge nTable is ≡
    containsUnacceptableEdge (λi j. nTable ⟦i,j⟧) is
```

5.6 Lower Bounds for the Total Weight

The definition of *tame* implies the existence of an admissible weight function $w : G \to \mathbf{N}$ with total weight $\sum_f w(f)$ less than the target, 14.8. We do not actually solve this linear optimization problem and calculate a weight assignment, but for a given final graph we calculate a lower bound on the total weight and neglect a generated graph if this lower bound is greater than the target. Hence we may also generate graphs that are not tame, but for the completeness of the enumeration it is sufficient that every tame plane graph is generated.

It is often possible to conclude from the examination of a partial plane graph g that for all plane graphs generated from g any admissible weight assignment will have a total weight greater than the target (14.8). In such cases, g can be neglected. Hence we are looking for lower bounds for the total weight $\sum_{f \in \mathit{faces}\ g'} w\ f$ in any graph g' generated from g, which can be derived from the following restrictions for an admissible weight assignment (properties $\mathit{admissible}_0$ - $\mathit{admissible}_4$, see Section 4.4).

1. Admissible weights are natural numbers. The weight $w\ f$ of a final f is at least $\mathbf{d}\ |\mathit{vertices}\ f|$ ($\mathit{admissible}_1$). Moreover, every final face f in a partial plane graph g will be present in any plane graph generated from g. Thus the total weight $\sum_{f \in \mathit{faces}\ g'} w\ f$ in any graph g' generated from g will be at least the sum $\sum_{f \in \mathit{finals}\ g} \mathbf{d}\ |\mathit{vertices}\ f|$ of the values of $\mathbf{d}$ of all final faces.

2. We can obtain better lower bounds for *final* vertices using the restrictions of $\mathit{admissible}_2$. If a final vertex v is not *exceptional* (i.e. none of the incident faces is exceptional), the sum $\sum_{f \in \mathit{facesAt}\ g\ v} w\ f$ of the weights of all incident faces is at least $\mathbf{b}\ (\mathit{tri}\ g\ v)\ (\mathit{quad}\ g\ v)$ ($\mathit{admissible}_2$).

3. For a *separated* set V of *final* vertices (no two vertices in V are adjacent or lie on a common quadrilateral and all vertices in V are exceptional), the sum $\sum_{f \in [f \in faces\ g.\ set\ (vertices\ f) \cap set\ V \neq \{\}]} w\ f$ of the weights of all faces incident with a vertex of V is at least $(admissible_4)$

$$\sum_{f \in [f \in faces\ g.\ set\ (vertices\ f) \cap set\ V \neq \{\}]} \mathbf{d}\ |vertices\ f| + \sum_{v \in V} \mathbf{a}\ (tri\ g\ v).$$

Combining these criteria, every *preseparated* (no two vertices in V are adjacent or lie on a common quadrilateral) set V of vertices gives rise to a lower bound.

We partition V (represented as list) in three subsets:

$$\begin{aligned} V1 &= [v \in V\ .\ except\ g\ v = 0] \\ V2 &= [v \in V\ .\ except\ g\ v \neq 0 \wedge degree\ g\ v = 5] \\ V3 &= [v \in V\ .\ except\ g\ v \neq 0 \wedge degree\ g\ v \neq 5] \end{aligned}$$

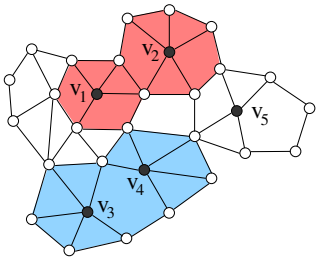
We partition the set of faces in g in:

$$\begin{aligned} F1 &= [f \in faces\ g\ .\ set\ (vertices\ f) \cap set\ V = \{\}] \\ F2 &= [f \in faces\ g\ .\ set\ (vertices\ f) \cap set\ V \neq \{\}] \end{aligned}$$

We calculate a lower bound by

$$\sum_{v \in V1} \mathbf{b}\ (tri\ g\ v)\ (quad\ g\ v) + \sum_{f \in F1} \mathbf{d}\ f + \sum_{v \in V2} \mathbf{a}\ (tri\ g\ v).$$

Example



From the preseparated set $V = \{v_1, v_2, v_3, v_4, v_5\}$, we calculate a lower bound for the total weight. The set of non-exceptional vertices V_1 is $\{v_1, v_2\}$. All faces incident with one of the vertices of V_1 contribute $\mathbf{b}(2,2) + \mathbf{b}(3,2)$ to the lower bound. The set of exceptional vertices of degree 5 is the separated set $V_2 = \{v_3, v_4\}$. The contribution of all faces incident with one of the vertices in V_2 is $8 * \mathbf{d}(3) + 2 * \mathbf{d}(4) + \mathbf{d}(5)$. All other faces contribute together $7 * \mathbf{d}(3) + 3 * \mathbf{d}(4) + \mathbf{d}(5)$. Hence, the lower bound is calculated as $\mathbf{b}(2,2) + \mathbf{b}(3,2) + 2 * \mathbf{a}(4) + 15 * \mathbf{d}(3) + 3 * \mathbf{d}(4) + 2 * \mathbf{d}(5)$. Of course, by far there exists no admissible weight assignment for this graph.

Lower Bounds for Final Graphs

Subsequently we calculate a lower bound for the total weight of an admissible weight function.

Instead of actually calculating a preseparated set V and partitioning it into V_1, V_3 and V_3, we first calculate a lower bound by $\sum_{f \in \mathit{finals}\ g}$ **d** $|\mathit{vertices}\ f|$ (by the function $\mathit{faceSquanderLowerBound}_{\mathit{param}}\ g$) and then add the *excess* $\mathit{ExcessNotAt}_{\mathit{param}}$, i.e. the maximal contribution of all faces to the lower bound that exceeds this sum. First, for every vertex v the *excess* at v is calculated (by the function $\mathit{ExcessAt}_{\mathit{param}}$), i.e. the contribution of the incident faces to the lower bound. Every preseparated set of vertices gives rise to a a lower bound for the total weight. The best (maximal) lower bound for all preseparated sets of vertices is calculated by the function $\mathit{ExcessNotAt}_{\mathit{param}}$.

faceSquanderLowerBound As a first lower bound for the total weight, the sum $\sum_{f \in \mathit{finals}\ g}$ **d** $|\mathit{vertices}\ f|$ is calculated by the function $\mathit{faceSquanderLowerBound}_{\mathit{param}}\ g$.

$\mathit{faceSquanderLowerBound} :: \mathit{parameter} \Rightarrow \mathit{graph} \Rightarrow \mathit{nat}$
$\mathit{faceSquanderLowerBound}_{\mathit{param}}\ g \equiv \sum_{f \in \mathit{finals}\ g}$ **d** $|\mathit{vertices}\ f|$

Excess Since for nonfinal vertices we cannot conclude better lower bounds, the excess of a nonfinal vertex is 0. The excess for a non-exceptional vertex of type (t, q) is the difference of the lower bounds given by $\mathbf{b'}_{\mathit{param}}\ t\ q$ and $t * \mathbf{d}\ 3 - q * \mathbf{d}\ 4$. This value is always positive, as can be checked by inspecting the definition of **b** and **d**. For an exceptional vertex of degree 5, the excess is the value **a** t.

```
excessAtType :: parameter ⇒ nat ⇒ nat ⇒ nat ⇒ nat
excessAtType_param t q e ≡
    if e = 0 then
        if 6 < t + q then squanderTarget
        else b'_param t q − t ∗ d 3 − q ∗ d 4
    else if t + q + e ≠ 5 then 0
        else a t

ExcessAt :: parameter ⇒ graph ⇒ vertex ⇒ nat
ExcessAt_param g v ≡ if ¬ finalVertex g v then 0
    else excessAtType_param (tri g v) (quad g v) (except g v)
```

For a given graph g and a given set of vertices v, the values of $ExcessAt_{param}$ g v are stored in a table $ExcessTable_{param}$ g vs, in order to avoid multiple calculations.

$ExcessTable :: parameter \Rightarrow graph \Rightarrow vertex\ list \Rightarrow (vertex \times nat)\ list$
$ExcessTable_{param}\ g\ vs \equiv$
$[(v,\ ExcessAt_{param}\ g\ v).\ v \in [v \in vs.\ 0 < ExcessAt_{param}\ g\ v\ ]]$

Lemma $(v,e) \in set\ (ExcessTable_{param}\ g\ (vertices\ g))$
$= (v \in set\ (vertices\ g)$
$\wedge\ finalVertex\ g\ v$
$\wedge\ 0 < ExcessAt_{param}\ g\ v$
$\wedge\ e = ExcessAt_{param}\ g\ v)$ $\surd$

Note that the table contains only entries for final vertices, since for nonfinal vertices the excess $ExcessAt_{param}$ g v is 0.

ExcessNotAt Every preseparated set gives rise to a lower bound for the total weight. Hence the best lower bound is obtained by the maximal lower bound of all separated sets V. The function *ExcessNotAt* calculates the greatest excess of all separated sets V.

The function *deleteAround* is used to construct a preseparated set of vertices. Given a vertex v and a list ps of pairs of vertices and excesses the function *deleteAround* removes all entries for vertices that are adjacent to v or lie on a common quadrilateral. It is based on the function *removeKeyList* (see Appendix A.1).

$deleteAround :: graph \Rightarrow vertex \Rightarrow (vertex \times nat)\ list \Rightarrow (vertex \times nat)\ list$
$deleteAround\ g\ v\ ps \equiv$
$let\ fs = facesAt\ g\ v;$
$ws = \cup_{f \in fs}\ if\ |vertices\ f| = 4\ then\ [f{\cdot}v,\ f^2{\cdot}v]\ else\ [f{\cdot}v]\ in$
$removeKeyList\ ws\ ps$

The function $ExcessNotAt_{param}$ g $(Some\ v)$ calculates the greatest excess for all separated sets V of vertices of g with the restriction that $v \in set\ V$, whereas $ExcessNotAt_{param}$ g $None$ calculates the greatest excess for all separated sets V of vertices of g without restrictions.

We calculate a table of excesses of all final vertices by the function $ExcessTable_{param}$ g $(vertices\ g)$. To calculate the maximal excess for all sets of preseparated vertices V that contain a vertex v, we remove all vertices from

the table which are adjacent to v or opposite to v in a quadrilateral. For all vertices $w \neq v$ in the table, we calculate the maximal excess for the case that $w \notin V$ or $w \in V$.

```
ExcessNotAtRec :: (nat, nat) table ⇒ graph ⇒ nat
ExcessNotAtRec [] g = 0
ExcessNotAtRec ((v,e)#ps) g =
    max (ExcessNotAtRec ps g)
        (e + ExcessNotAtRec (deleteAround g v ps) g)

ExcessNotAt :: parameter ⇒ graph ⇒ vertex option ⇒ nat
ExcessNotAt_param g v-opt ≡
    let ps = ExcessTable_param g (vertices g) in
    case v-opt of None ⇒  ExcessNotAtRec ps g
        | Some v ⇒ ExcessNotAtRec (deleteAround g v ps) g
```

Lemma $g \in PlaneGraphs_1 \Longrightarrow$
$\exists V.\ ExcessNotAt_{param}\ g\ None = \sum_{v \in V} ExcessAt_{param}\ g\ v$
$\wedge\ preSeparated\ g\ (set\ V) \wedge set\ V \subseteq set\ (vertices\ g)$
$\wedge\ distinct\ V$ $\surd$

Finally, a lower bound for a final graph can be calculated by the sum of $\sum_{f \in finals\ g}$ **d** $|vertices\ f|$ and the excess.

```
squanderLowerBound :: parameter ⇒ graph ⇒ nat
squanderLowerBound_param g ≡
    faceSquanderLowerBound_param g + ExcessNotAt_param g None
```

Lower Bounds for Nonfinal Graphs

Subsequently, we derive lower bounds for nonfinal graphs. We first calculate the set of admissible types *admissibleTypes*. Then we calculate for a nonfinal vertex the least reachable lower bound using the function *squanderForecast*. The function *neglectableModification* is used to determine whether a modification of a graph can be neglected, since the lower bound is exceeded. Finally, the function *neglectableVertexList* checks if the modification of a graph by an *addFace* operation with a vertex list *vs* is neglectable at one of the vertices of *vs*.

admissibleTypes The function *admissibleTypes* calculates all types (p, q) of vertices where **b'**$_{param}\ p\ q < squanderTarget$.

$squanderVertexLength :: nat$
$squanderVertexLength \equiv 7$

$admissibleTypes :: parameter \Rightarrow (nat \times nat)\ list$
$admissibleTypes_{param} \equiv let\ Q = [0\ ..<\ squanderVertexLength]\ in$
$[(p,q) \in Q \times Q.\ \mathbf{b'}_{param}\ p\ q < squanderTarget]$

The correctness theorem for $admissibleTypes_{param}$ can be stated as follows:

Lemma $\mathbf{b'}_{param}\ t\ q < squanderTarget \Longrightarrow$
$0 \leq t \wedge t < squanderVertexLength \wedge 0 \leq q \wedge q < squanderVertexLength\ \surd$

Lemma
$(t,q) \in set\ (admissibleTypes_{param}) = (\mathbf{b'}_{param}\ t\ q < squanderTarget)$ $\surd$

squanderForecast Since a nonfinal vertex v in a graph g does not contribute to the calculated excess, we further improve the lower bound by considering all graphs g' that may possibly be generated from g. We calculate the minimal reachable lower bound of the weights of all incident faces (that can be obtained by property $admissible_2$). More precisely, for a non-exceptional vertex v we calculate a lower bound of the contribution $\mathbf{b'}_{param}$ $(tri\ g'\ v)$ $(quad\ g'\ v)$ of the incident faces for all graphs g' generated from g, if v is also not an exceptional in g'. Moreover, implicitly the following property of the constants **b**, **d**, and **a** is used: if v is exceptional in g', the contribution to the lower bound would even be higher.

Assume that v is incident with $t = tri\ g\ v$ final triangles, $q = quad\ g\ v$ final quadrilaterals and $temp$ nonfinal faces. Since final faces are preserved, in every final graph g' generated from g, the number of incident triangles $t' = tri\ g'\ v$ must be at least t and the number of incident quadrilaterals $q' = quad\ g'\ v$ must be at least q. Moreover every nonfinal face may be replaced by at least one triangle or quadrilateral. Hence the lower bound $\mathbf{b'}_{param}$ $(tri\ g'\ v)$ $(quad\ g'\ v)$ for all graphs g' generated from g can be calculated as

$$min\{\mathbf{b}(t',q').\quad \begin{array}{l}\mathbf{b'}_{param}\ p'\ q' < squanderTarget \wedge \\ tri\ g\ v \leq t' \wedge quad\ g\ v \leq q' \wedge \\ tri\ g\ v + quad\ g\ v + temp \leq t' + q'\ \}.\end{array}$$

We calculate the lower bound $squanderForecast$ for $\mathbf{b'}_{param}$ $(tri\ g'\ v)$ $(quad\ g'\ v)$ for all graphs g' generated from g:

$squanderForecast :: parameter \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$

$$
\begin{array}{l}
\mathit{squanderForecast}_{\mathit{param}}\ t\ q\ \mathit{temp} \equiv \\
\quad \mathit{minList} \\
\quad\quad \mathit{squanderTarget} \\
\quad\quad [\mathbf{b'}_{\mathit{param}}\ t'\ q'.\ (t',q') \in [(t',q') \in \mathit{admissibleTypes}_{\mathit{param}}. \\
\qquad\qquad\qquad\qquad\qquad\qquad t + q + \mathit{temp} \leq t' + q' \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ t \leq t' \wedge\ q \leq q']]
\end{array}
$$

If the vertex is final, then the value is calculated exactly.

Lemma $\mathit{squanderForecast}_{\mathit{param}}\ t\ q\ 0 = \mathbf{b'}_{\mathit{param}}\ t\ q$ *!*

To avoid duplicate calculations, we the store calculated results of the lower bounds $\mathbf{b'}_{\mathit{param}}\ (\mathit{tri}\ g'\ v)\ (\mathit{quad}\ g'\ v)$ in a table *squanderForecastTable*.

$$
\begin{array}{l}
\mathit{squanderForecastTable} :: \mathit{parameter} \Rightarrow \mathit{nat\ array\ array\ array} \\
\mathit{squanderForecastTable}_{\mathit{param}} \equiv \\
\quad \mathit{let}\ l = \mathit{squanderVertexLength}\ \mathit{in} \\
\quad [\![\mathit{squanderForecast}_{\mathit{param}}\ t\ q\ \mathit{temp}.\ t < l,\ q < l,\ \mathit{temp} < l]\!]
\end{array}
$$

neglectableModification The function *neglectableModification* checks if a modification of a graph g at a vertex v leads to a graph g' and the lower bound for g' already exceeds the target. If this is the case, then the modification can be neglected. The arguments are t_n, the number of new triangles; q_n, the number of new quadrilaterals; e_n, the size of a new exceptional face if there is one, otherwise 0; and temp_n, the number of new nonfinal faces (which can also be negative). We calculate for the new graph g' the numbers of triangles, quadrilaterals exceptionals and nonfinal faces at vertex v:

$$
\begin{array}{l}
t' = \mathit{tri}\ g'\ v = \mathit{tri}\ g\ v + t_n, \\
q' = \mathit{quad}\ g'\ v = \mathit{quad}\ g\ v + q_n, \\
e' = \mathit{except}\ g'\ v = \mathit{except}\ g\ v + e_n, \\
\mathit{temp}' = \mathit{temp}_n + |\mathit{nonFinalsAt}\ g\ v|.
\end{array}
$$

A graph g' is excluded if the degree of a vertex exceeds the bounds given by the properties tame_4 and tame_5. Moreover we calculate the lower bound for the total weight for g'. All new final faces of length n in g' will contribute with $\mathbf{d}\ n$, hence this value is added to the current lower bound. If v is not exceptional in g' (which is the precondition for the calculation of *squanderForecast*), then we calculate a new excess at vertex v. The lower bound is calculated for all separated sets V that do not contain v, and for those that contain v. The modification is neglectable if the lower bound of the total weight exceeds the target.

$$
\begin{aligned}
&\mathit{neglectableModification} :: \\
&\quad \mathit{parameter} \Rightarrow \mathit{graph} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{int} \Rightarrow \mathit{vertex} \Rightarrow \mathit{bool} \\
&\mathit{neglectableModification}_{\mathit{param}}\ g\ t_n\ q_n\ e_n\ \mathit{temp}_n\ v \equiv \\
&\quad \mathit{let}\ t' = \mathit{tri}\ g\ v + t_n; \\
&\quad q' = \mathit{quad}\ g\ v + q_n; \\
&\quad \mathit{temp}' = \mathit{nat}\ (\mathit{int}\ |\mathit{nonFinalsAt}\ g\ v| + \mathit{temp}_n); \\
&\quad e' = \mathit{except}\ g\ v + (\mathit{if}\ 0 < e_n\ \mathit{then}\ 1\ \mathit{else}\ 0)\ \mathit{in} \\
&\quad 0 < e' \wedge 5 < t' + q' + \mathit{temp}' + e' \\
&\quad \vee\ \ e' = 0 \wedge 6 < t' + q' + \mathit{temp}' \\
&\quad \vee\ \mathit{squanderTarget} \leq \mathit{faceSquanderLowerBound}_{\mathit{param}}\ g \\
&\qquad\qquad + (\mathit{if}\ 0 < e_n\ \mathit{then}\ \mathbf{d}\ e_n\ \mathit{else}\ 0) \\
&\qquad\qquad + t_n * \mathbf{d}\ 3 \\
&\qquad\qquad + q_n * \mathbf{d}\ 4 \\
&\qquad\qquad + \mathit{ExcessNotAt}_{\mathit{param}}\ g\ \mathit{None} \\
&\quad \vee\ e' = 0 \\
&\qquad \wedge\ \mathit{squanderTarget} \leq \mathit{faceSquanderLowerBound}_{\mathit{param}}\ g \\
&\qquad\qquad + (\mathit{if}\ 0 < e_n\ \mathit{then}\ \mathbf{d}\ e_n\ \mathit{else}\ 0) \\
&\qquad\qquad + t_n * \mathbf{d}\ 3 \\
&\qquad\qquad + q_n * \mathbf{d}\ 4 \\
&\qquad\qquad + \mathit{ExcessNotAt}_{\mathit{param}}\ g\ (\mathit{Some}\ v) \\
&\qquad\qquad + (\mathit{squanderForecastTable}_{\mathit{param}}\ [\![t', q', \mathit{temp}']\!] \\
&\qquad\qquad\quad - t' * \mathbf{d}\ 3 \\
&\qquad\qquad\quad - q' * \mathbf{d}\ 4)
\end{aligned}
$$

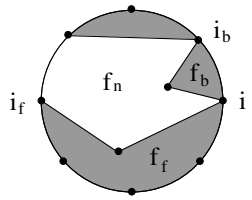
neglectableVertexList The function *neglectableVertexList vs f g* decides if a modification of a graph g by *addFace* can be neglected, if a new final face with vertex list *vs* is added in a nonfinal face f.

neglectableVertexList vs f g checks if the modification is neglectable at one of the vertices of the new face whereas $\mathit{neglectableAtVertex}_{\mathit{param}}\ i\ \mathit{vs}\ f\ g$ checks if the modification is neglectable at the ith vertex in the list *vs*.

We calculate for the new graph g' the numbers of *new* triangles t_n, the number of *new* quadrilaterals q_n, the length e_n of a *new* exceptional (if one exists), and the number of *new* nonfinal faces temp_n at vertex v.

The *addFace* operation creates at every vertex *Some v* in *vs* the new final face f_n of the length of *vs*, a nonfinal face f_f of length n_f in forward direction of face f from v, and a nonfinal face f_b of length n_b in backward direction of face f_f from v. If one of the nonfinal faces is a triangle, we may assume that this face is made final (property tame_2) and hence the number of new final triangles can be increased by 1. Moreover for every nonfinal face of length

at least 3 the number of new nonfinal faces is increased by 1.



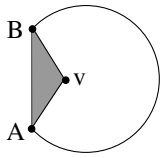
Finally we check if the modification is neglectable.

```
succeedingNulls :: nat ⇒ vertex option list ⇒ nat
succeedingNulls n [] = n
succeedingNulls n (v#vs) =
    (case v of None ⇒ succeedingNulls (n + 1) vs
        | Some w ⇒ n)

neglectableAtVertex ::
    parameter ⇒ nat ⇒ vertex option list ⇒ face ⇒ graph ⇒ bool
neglectableAtVertex_param  i vs f g ≡
    case vs⟦i⟧ of None ⇒ False
    | Some v ⇒

    (* the new final face of length n *)
    let n = length vs;
    temp_n = −1;
    e_n = (if 4 < n then n else 0);
    q_n = (if n = 4 then 1 else 0);
    t_n = (if n = 3 then 1 else 0);

    (* the nonfinal face in forward direction *)
    (* i_f is the index of the first non null vertex in direction of face f *)
    i_f = (i + 1) mod n;
    i_f = (i_f + succeedingNulls 0 (drop i_f vs @ take i_f vs)) mod n;
    n_f = directedLength f (the vs⟦i⟧) (the vs⟦i_f⟧)
          + nat ((int i_f − int i) mod int n);
    t_n = t_n + (if n_f = 3 then 1 else 0);
    temp_n = temp_n + (if 3 < n_f then 1 else 0);

    (* the nonfinal face in backward direction *)
    (* i_b is the index of the first non null vertex in opposite direction of face f *)
    i_b = (i − 1) mod n;
    i_b = (i_b − succeedingNulls 0 (rev (drop i vs @ take i vs))) mod n;
    n_f = directedLength f (the vs⟦i_b⟧) (the vs⟦i⟧)
```

```
        + nat ((int i − int i_b) mod int n);
    t_n = t_n + (if n_f = 3 then 1 else 0);
    temp_n = temp_n + (if 3 < n_f then 1 else 0) in
    neglectableModification_param g t_n q_n e_n temp_n v

neglectableVertexList :: parameter ⇒ vertex option list ⇒ face ⇒ graph ⇒ bool
neglectableVertexList_param vs f g ≡
    ∃ i ∈ set [0..<|vs| ]. neglectableAtVertex_param i vs f g
```

5.7 Forced Triangles

It is often possible to conclude from the examination of a nonfinal graph at a particular vertex v, that the only possibility to obtain a tame graph is to replace a nonfinal face at v by a triangle. We then say that a triangle is *forced* at the vertex v. This can be proved by verifying that turning it into any other polygon or combination of polygons would exceed the target weight, but changing it to a triangle would not.



We first calculate a lower bound for the weight of a new face of length at least *ngon* by the function *squanderFaceStartingAt*. The calculated lower bound $\sum_{f \in \mathit{finals}\ g}$ **d** $|\mathit{vertices}\ f|$ (from $\mathit{admissible}_1$) would be increased by this value if a nonfinal face at vertex v is replaced by some face of length at least *ngon*.

```
squanderFaceStartingAt :: parameter ⇒ nat ⇒ nat
squanderFaceStartingAt_param ngon ≡
    case param of QuadParameter n => d ngon
        | ExceptionalParameter m =>
            minList squanderTarget [d i. i ∈ [ngon..toNat m]]
```

Now, we determine if a triangle is forced at a vertex v in a graph g by the function *ForcedTriangleAt*.

Consider the first case that there are no exceptional final faces incident with v (*except* $g\ v = 0$). A triangle is forced at a vertex v in a graph g if

- no exceptional face can be created at vertex v (otherwise the lower bound of the total weight, calculated by $\sum_{f \in \mathit{finals}\ g}$ **d** $|\mathit{vertices}\ f|$ exceeds the target) and
- the nonfinal face can neither be replaced by one quadrilateral
- nor by two triangles.

The latter cases are excluded using $\mathit{squanderForecast}_{\mathit{param}}$ (which is the minimal reachable value of **b** $t\ q$ at a nonfinal vertex) to calculate a lower bound for the weights of the incident faces (using property $\mathit{admissible}_2$). The function implicitly uses the following property of the function **b**: provided we can add one triangle at v but not one quadrilateral (then **b** $(t + 1)\ q \leq$ **b** $t\ (q + 1)$), then we can neither add one quadrilateral and some additional faces (because **b** $t\ (q + 1) \leq$ **b** $(t + i)\ (q + 1 + j)$ for all i, j) nor one triangle and some additional faces (because **b** $(t + 2)\ q \leq$ **b** $(t + 2 + i)\ (q + j)$ for all i, j). Hence all other cases than adding a single triangle at v are excluded.

Note that the lower bound used by the function could also be increased by $\mathit{ExcessNotAt}_{\mathit{param}}\ g\ (\mathit{Some}\ v)$, which would lead to more forced triangles.

For the second case that there are exceptional faces incident with v ($\mathit{except}\ g\ v \neq 0$), a triangle is forced at a vertex v in a graph g, if

- no quadrilateral or exceptional face can be created at v (by $\mathit{admissible}_1$) and
- the nonfinal face cannot be replaced by two or more faces (otherwise the degree bound 5 of an exceptional vertex (tame_5) would be exceeded).

$\mathit{ForcedTriangleAt} :: \mathit{parameter} \Rightarrow \mathit{graph} \Rightarrow \mathit{vertex} \Rightarrow \mathit{bool}$
$\mathit{ForcedTriangleAt}_{\mathit{param}}\ g\ v \equiv$
  $\mathit{let}\ t = \mathit{tri}\ g\ v;$
  $q = \mathit{quad}\ g\ v;$
  $\mathit{tempX} = |\mathit{nonFinalsAt}\ g\ v|;$
  $e = \mathit{except}\ g\ v;$
  $\mathit{fsq} = \mathit{faceSquanderLowerBound}_{\mathit{param}}\ g;$
  $\mathit{fsqred} = \mathit{fsq} - q * \mathbf{d}\ 4 - t * \mathbf{d}\ 3;$
  $\mathit{target} = \mathit{squanderTarget};$
  $\mathit{excessNot} = \mathit{ExcessNotAt}_{\mathit{param}}\ g\ (\mathit{Some}\ v)\ \mathit{in}$
  $\mathit{if}\ (e = 0)$

```
  then fsq + squanderFaceStartingAt_param 5 > target
     ∧ squanderForecast_param t (q + 1) (tempX − 1)
         + fsqred + excessNot > target
     ∧ squanderForecast_param (t + tempX + 1) q 0
         + fsqred + excessNot > target

  else let nextface = squanderFaceStartingAt_param 4 in
     fsq + excessNot + nextface > target
    ∧ t + tempX + q + e + 1 > 5
```

If a triangle is forced at a vertex v in a graph g, we can modify the usual successor function for a graph g and follow only the path in the tree where a triangle is constructed at vertex v. The graph operation *handleForcedTriangle* adds a forced triangle at a vertex v. The result is *None* if there is no forced triangle in g. If there is a forced triangle in g, where A and B are the neighbors of v in a nonfinal face f of g, but the modification of adding a triangle at v is neglectable or the edge (a, b) is neglectable, the result is *Some* []. Otherwise we construct a new graph g' by adding the nonfinal triangle and return *Some* g'.

```
handleForcedTriangleVertex ::
    parameter ⇒ graph ⇒ vertex list ⇒ graph list option
handleForcedTriangleVertex_param g [] = None
handleForcedTriangleVertex_param g (v#vs) =
    (let nF = nonFinalsAt g v in
    if nF = [] then handleForcedTriangleVertex_param g vs      (* continue *)
    else if ¬ ForcedTriangleAt_param g v
    then handleForcedTriangleVertex_param g vs                 (* continue *)
    else let f = hd nF; A = f•v; B = f^-1•v;
    g' = addFace g f [Some v, Some A, Some B] in
    if neglectableModification_param g 1 0 0 −1 A then Some []
    else if neglectableModification_param g 1 0 0 −1 B then Some []
    else if neglectableEdge g f A B then Some []
    else Some [g'])

handleForcedTriangle :: parameter ⇒ graph ⇒ graph list option
handleForcedTriangle_param g ≡
    handleForcedTriangleVertex_param g (vertices g)
```

This refinement step is again an optimization step that does not reduce the set of generated graphs modulo isomorphism. It is an optimization in the sense that it reduces the search space. To prove correctness it must be shown that we do not lose tame graphs by introducing this optimization. We prove

the correctness of this refinement step by executing the algorithm without this refinement (as in Section 6.5). Executing the enumeration without this optimization does not even lead to a slow-down of the algorithm, since the test to determine if there exists a forced triangle in a graph also requires some complex calculations.

Hence we remove this optimization from the definition and simplify the proof of the completeness of the enumeration algorithm.

5.8 Nonfinal Triangles and Quadrilaterals

Nonfinal Triangles

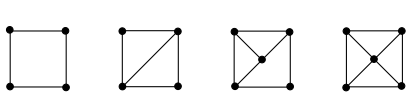
In a tame graph every triangle is either a face or the opposite of a face (property $tame_2$). Hence every nonfinal triangle can be made final (except for the initial triangle graph). The function *makeTrianglesFinal* makes all nonfinal faces in a graph final. It repeatedly applies the function *makeFaceFinal* to all triangles in the graph.

$triangle :: face \Rightarrow bool$
$triangle\ f \equiv |vertices\ f| = 3$

makeTrianglesFinal:
$makeTrianglesFinal\ g \equiv foldr\ makeFaceFinal\ [f \in faces\ g.\ triangle\ f]\ g$

Nonfinal Quadrilaterals

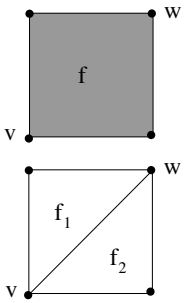
A nonfinal quadrilateral surrounds one of the tame configurations for quadrilaterals (property $tame_3$).



Hence a nonfinal quadrilateral can be replaced by all possible configurations.

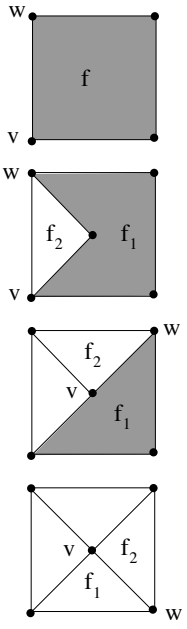
The operation *splitQuad* splits a nonfinal face f at vertex v in two final triangles, provided this modification of the graph is not neglectable at one of the vertices of f.

```
splitQuad :: parameter ⇒ vertex ⇒ face ⇒ graph ⇒ graph list
splitQuad_param v f g ≡
    if |vertices f| ≠ 4 ∨ final f then []
    else let w = f^2•v in
    if ∃ i ∈ set [0..3]. neglectableModification_param g
        (1 + ((i + 1) mod 2)) 0 0 −1 (f^i•v)
    then []
    else let (f_1, f_2, g) = FaceDivisionGraph g w v f [];
    g = makeFaceFinal f_1 g;
    g = makeFaceFinal f_2 g in
    [g]
```



The operation *Do40* creates a vertex of type $(4, 0)$ in the interior of a nonfinal face f, provided this modification is not neglectable at one of the vertices of f and the excess of the new vertex does not lead to a lower bound exceeding the target.

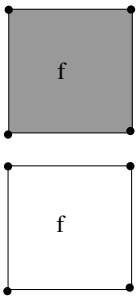
```
Do40 :: parameter ⇒ face ⇒ graph ⇒ graph list
Do40_param f g ≡
    if |vertices f| ≠ 4 ∨ final f then []
    else let faceSqu = faceSquanderLowerBound_param g;
        exN = ExcessNotAt_param g None;
        forecast = squanderForecast_param 4 0 0 in
    if squanderTarget ≤ faceSqu + exN + forecast then []
    else if ∃ v ∈ set (vertices f).
        neglectableModification_param g 2 0 0 −1 v then []
    else let vs = [countVertices g];
        v = (vertices f)⟦0⟧; w = f•v;
        (f_1, f_2, g) = FaceDivisionGraph g w v f vs ;
        g = makeFaceFinal f_2 g;
        v = f_1•v; w = f_1•w;
        (f_1, f_2, g) = FaceDivisionGraph g w v f_1 [];
        g = makeFaceFinal f_2 g;
        w = f_1• w;
        (f_1, f_2, g) = FaceDivisionGraph g w v f_1 [];
        g = makeFaceFinal f_1 g;
        g = makeFaceFinal f_2 g in
        [g]
```



The operation *makeQuadFinal* makes a nonfinal quadrilateral final, provided

this modification of the graph is not neglectable at one of the vertices of f.

```
makeQuadFinal :: parameter ⇒ face ⇒ graph ⇒ graph list
makeQuadFinal_param f g ≡
    if |vertices f| ≠ 4 ∨ final f then []
    else if ∃ v ∈ set (vertices f).
        neglectableModification_param g 0 1 0 −1 v
    then []
    else [makeFaceFinal f g]
```



We define a function *handleQuad* that calculates possible successor graphs and a function *isQuadFriendly* to decide if we can apply the function *handleQuad* to a graph.

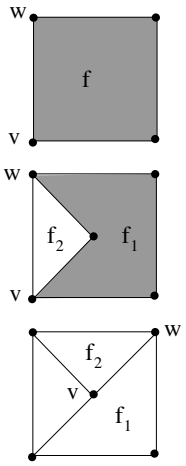
We may not apply the function *handleQuad* if a graph g contains less than 6 vertices, because g could just consist of one of the tame configurations bounded by a nonfinal quadrilateral. Moreover, for efficiency reasons we may decide to call the usual successor function if a vertex of type $(2, 1)$ can be added to the graph without exceeding the weight target. We obtain the proof obligation that whenever *isQuadFriendly* is *True*, no other successors than the graphs generated by the function *handleQuad* are possible.

```
isQuadFriendly_(QuadParameter n) g =
    (let lb = squanderLowerBound_(QuadParameter n) g in
    countVertices g ≥ 6
    ∧ (lb + b'_(QuadParameter n) 2 1 ≥ squanderTarget))
isQuadFriendly_(ExceptionalParameter n) g =
    (let lb = squanderLowerBound_(ExceptionalParameter n) g in
    countVertices g ≥ 6
    ∧ (lb + squanderFaceStartingAt_(ExceptionalParameter n) 5 ≥ squanderTarget)
    ∧ (lb + b'_(ExceptionalParameter n) 2 1 ≥ squanderTarget))
```

```
handleQuad_param f g ≡ if |vertices f| ≠ 4 then []
    else let vs = vertices f; v = hd vs; n = f·v in
    (splitQuad_param n f g)
    @(splitQuad_param v f g)
    @(Do40_param f g)
    @(makeQuadFinal_param f g)
```

Alternatively, we could use a function *handleQuad2* which additionally calls an operation *Do21*, creating a vertex of type $(2, 1)$ in the interior of a nonfinal face.

```
Do21 :: parameter ⇒ vertex ⇒ face ⇒ graph ⇒ graph list
Do21 param v f g ≡
    if |vertices f| ≠ 4 ∨ final f then []
    else let lb = squanderLowerBound_param g in
    if squanderTarget ≤ lb + b'_param 2 1 then []
    else if (∃ i ∈ set [0..2].
            neglectableModification_param g 2 0 0 −1 (f^i•v)
          ∨ neglectableModification_param g 1 0 0 −1 (f^3•v))
    then []
    else let vs = [countVertices g];
        w = f•v;
        (f_1, f_2, g) = FaceDivisionGraph g w v f vs ;
        g = makeFaceFinal f_2 g;
        v = f_1•v; w = f_1•w;
        (f_1, f_2, g) = FaceDivisionGraph g w v f_1 [];
        g = makeFaceFinal f_1 g;
        g = makeFaceFinal f_2 g in
        [g]
```



The alternative function creates all possible tame configurations for quadrilaterals.

```
handleQuad2-def: handleQuad2_param f g ≡
    if |vertices f| ≠ 4 then []
    else let vs = vertices f;
    v = hd vs; n = f•v; n_2 = f^2•v;  n_3 = f^3•v in
    (splitQuad_param n f g)
    @(splitQuad_param v f g)
    @(Do21_param v f g)
    @(Do21_param n f g)
    @(Do21_param n_2 f g)
    @(Do21_param n_3 f g)
    @(Do40_param f g)
    @(makeQuadFinal_param f g)
```

We show by execution that both definitions generate the same set of graphs modulo isomorphism.

5.9 Neglectable Nonfinal Graphs

A nonfinal graph is neglected in the following cases:

- For a nonfinal graph g, we calculate a lower bound for the total weight $\sum_{f \in finals\ g'} w\ |vertices\ f|$ of all graphs g' that may be generated from g. A graph can be neglected if this lower bound exceeds 14.8 ($tame_7$) (calculated by *neglectableVertexList*). For the calculation of the lower bound see Section 5.6.
- A new generated (nonfinal) graph is neglected if a 3-cycle is created with enclosed vertices (calculated by *neglectableEdge*). This would contradict property $tame_2$.
- Using property $tame_2$, the successor function can be simplified such that it replaces all nonfinal triangles by final ones (*makeTrianglesFinal*). If a graph contains a nonfinal quad, then it can be replaced by all configurations that are allowed by property $tame_3$ (*handleQuads*).

```
generatePolygonOpt :: parameter ⇒ nat ⇒ vertex ⇒ face ⇒ graph ⇒ graph list
generatePolygonOpt_param ngon v f g ≡
    let neglectTable = neglectableEdgeTable g f v;
    enumeration = enumerator ngon |vertices f|;
    enumeration = [is ∈ enumeration.
        ¬ containsNeglectableEdge neglectTable is];
    vertexLists = [indexToVertexList f v is. is ∈ enumeration];
    vertexLists = [vs ∈ vertexLists. ¬ neglectableVertexList_param vs f g] in
    [addFace g f vs. vs ∈ vertexLists]
```

The function *polyLimit* calculates the maximal length of a face that can be added without exceeding the lower bound on the total weight.

```
polyLimit_param g ≡
    let lb = squanderLowerBound_param g in
    maxList 3 [n ∈ [3 .. maxGon_param]. lb + d n ≤ squanderTarget]
```

The successor function additionally excludes graphs that contain an earlier seed graph and that are neglectable by base point symmetry.

```
successorsListOpt :: parameter ⇒ graph ⇒ graph list
successorsListOpt_param g ≡
    if final g then []
```

```
  else let fopt = minimalTempFace g in
    case fopt of None ⇒ []
     | Some f ⇒
       if |vertices f| = 4 ∧ isQuadFriendly_param g
       then [makeTrianglesFinal g'. g' ∈ (handleQuad_param f g)]
       else let v = minimalVertex g f;
       polylimit = polyLimit_param g;
       rs = rev (∪_{i ∈ [3..polylimit]} generatePolygonOpt_param i v f g) in
       [makeTrianglesFinal g. g ∈ rs]


succsPlane_8 :: parameter ⇒ graph ⇒ graph list
succsPlane_8 param g ≡
  [g' ∈ successorsListOpt_param g.
    ¬ neglectableByBasePointSymmetry g'
   ∧ ¬ containsEarlierSeed_param g']

PlaneGraphs_8 :: graph set
PlaneGraphs_8 ≡ terminalsTree (Seed::parameter ⇒ graph) succsPlane_8

PlaneGraphs_8Param :: parameter ⇒ graph set
PlaneGraphs_8param ≡ terminalsTree_param Seed succsPlane_8

PlaneGraphs_8Tree :: parameter ⇒ graph set
PlaneGraphs_8Tree_param ≡ Tree succsPlane_8 param Seed_param
```

For correctness theorem is expressed as follows (see Section 6.9.

Theorem *plane*$_8$*-complete*:
$g \in \mathit{PlaneGraphs}_7 \Longrightarrow \mathit{tame}\ g \Longrightarrow g \in_{\cong} \mathit{PlaneGraphs}_8$

5.10 Neglectable Final Graphs

A final graph can be neglected in the following cases:

- A graph g can be neglected, if the lower bound of the total weight $\sum_{f \in \mathit{faces}\ g} w\ |\mathit{vertices}\ f|$ for an admissible weight assignment w exceeds 14.8 (tame_7). For the calculation of the lower bound see Section 5.6.
- A graph can be neglected if $\sum_f c(|f|) < 8$ (tame_6).

- A graph can be neglected if it contains vertices whose degree is too high ($tame_4$, $tame_5$).
- A graph can be neglected if it contains one of the forbidden configurations, i.e. two adjacent vertices of type $(4, 0)$ ($tame_8$) or a vertex with one incident triangle and one incident pentagon ($tame_3$, $tame_6$).

We calculate the total score of a graph.

```
scoreTarget :: nat
scoreTarget ≡ 8000

scoreUpperBound :: graph ⇒ nat
scoreUpperBound g ≡ nat (∑ f∈finals g c |vertices f| )
```

The function *has101Type* determines whether a graph contains a vertex with one incident triangle and one incident pentagon.

```
vertexHas101Type :: graph ⇒ vertex ⇒ bool
vertexHas101Type g v ≡
  (if ¬ finalVertex g v then False
  else if degree g v ≠ 2 then False
  else |vertices (facesAt g v [[0]] )| = 3 ∧ |vertices (facesAt g v [[1]])| = 5
     ∨ |vertices (facesAt g v [[0]])| = 5 ∧ |vertices (facesAt g v [[1]])| = 3)

has101Type :: graph ⇒ bool
has101Type g ≡ ∃ v ∈ set (vertices g). vertexHas101Type g v
```

The function *neglectableFinal* summarizes the conditions for a final graph to be neglectable: if the degree bounds are too high, the lower bound for the total weight exceeds the target, or the total score is less than 8.

```
neglectableFinal :: parameter ⇒ graph ⇒ bool
neglectableFinal_param g ≡
   final g
   ∧ ((∃ v ∈ set (vertices g). except g v = 0 ∧ 6 < degree g v
        ∨ 0 < except g v ∧ 5 < degree g v)
     ∨ squanderTarget ≤ squanderLowerBound_param g
```

$\vee$ *scoreUpperBound* $g <$ *scoreTarget*
$\vee$ *has101Type* g
$\vee$ *hasAdjacent40* g)

Finally we define the function *succsEnumeration*, by excluding all neglectable final graphs from $PlaneGraphs_8$.

succsEnumeration :: *parameter* $\Rightarrow$ *graph* $\Rightarrow$ *graph list*
$succsEnumeration_{param}\ g \equiv$
$[g' \in successorsListOpt_{param}\ g.\ \neg\ neglectableFinal_{param}\ g'$
$\wedge\ \neg\ neglectableByBasePointSymmetry\ g'\ ]$

Enumeration :: *graph set*
Enumeration $\equiv$ *terminalsTree Seed succsEnumeration*

EnumerationParam :: *parameter* $\Rightarrow$ *graph set*
$EnumerationParam\ param \equiv terminalsTree_{param}\ Seed\ succsEnumeration$

EnumerationTree :: *parameter* $\Rightarrow$ *graph set*
$EnumerationTree\ param \equiv Tree\ succsEnumeration_{param}\ Seed_{param}$

The completeness theorem of this last refinement step is formally proved in Section 6.10.

Theorem *Enumeration-complete*:
$g \in PlaneGraphs_8 \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} Enumeration$

Chapter 6

Completeness of Enumeration

In the previous chapter we described an executable definition of *Enumeration*, a superset of the set of tame plane graphs. Since the enumeration is obtained from the definition of plane graphs by a sequence of refinement steps, which reduce the set of generated graphs. The completeness proof of this enumeration is obtained by the completeness of each step.

In order to prove these completeness theorems, we need invariants of the graph representation. Due to the inductive definition, we can reach a graph by many different constructions, if we add faces in a different order. Since we do not want to distinguish these graphs, we need specific lemmas how the faces in a construction can be reordered to obtain an isomorphic graph.

6.1 Reorderings of Faces

Some of the refinement steps described in Chapter 5 exclude graphs if an isomorphic graph is generated by another path. These refinements are optimizations in the sense that they reduce the number of generated graphs, but modulo isomorphism the set of generated graphs is not reduced.

An example is the optimization used in our alternative definition of plane graphs, i.e. selecting a fixed edge in a nonfinal face and adding a new final face only at this edge (see Section 5.1). Another example is the optimization of neglecting nonfinal graphs containing a final vertex which is isomorphic to an earlier seed graph (see Section 5.3).

For the correctness proof of these optimizations it is essential to show that the order in which faces are created in the construction of a graph may be changed. However, not any arbitrary order of creation of faces is possible. For example, the constructed graph must be connected at any time and there are never two adjacent nonfinal faces in a partial graph.

Succeedingly, we describe two approaches for the proof of the correctness of such optimizations.

Proof by Execution

In some cases, we may prove an optimization by execution. We define an unoptimized enumeration function $Enumeration_{unopt}$ by modifying the original optimized definition *Enumeration* such that the optimization is removed from the algorithm. By construction, *Enumeration* generates a subset of $Enumeration_{unopt}$.

Lemma
$Enumeration \subseteq Enumeration_{unopt}$

We obtain a reduced archive *Archive* ([28]) from executing *Enumeration* and removing all graphs from the original archive which are not constructed by *Enumeration*. By construction

Lemma
$Enumeration =_{\cong} Archive$

Then we execute the function $Enumeration_{unopt}$ and compare the result with the reduced archive *Archive*: If we can verify

Lemma
$Enumeration_{unopt} \subseteq_{\cong} Archive$

by execution, we obtain a proof of the correctness of this optimization:

$Enumeration_{unopt} =_{\cong} Enumeration$

Then we may remove the optimization from the definition and simplify the correctness proof.

Of course this generally will lead to a slow-down of performance, but this is unimportant for the completeness proof. However once we have shown

that the same set of graphs is generated, we may even use the optimized algorithm.

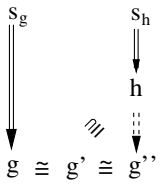
On the other hand, if the test does not succeed this does not necessarily mean that the optimization is incorrect. There is no complete test of tameness of a graph included in the algorithm. Graphs are only neglected if they are definitely not tame, but there may be some graphs generated which are not tame. Hence it may be the case that a graph which is not tame is filtered out in *Enumeration*, but not in $Enumeration_{unopt}$ because its faces were constructed in a different order. In order to leave out such an optimization, it is necessary to provide a complete test of tameness.

Reordering of Faces

We prove the following lemma about reordering of faces.

Lemma
If a plane graph h is a subgraph (modulo isomorphism) of a plane graph g then h can be extended to g (up to isomorphism).

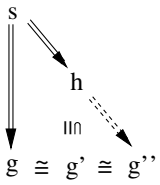
The condition of the lemma means that there is a graph g' isomorphic to g (obtained from g by a renaming φ) and h is a subgraph of g', i.e. *set* (*finals* h) $\subseteq$ *set* (*finals* g'). The conclusion means that there is a graph g'', isomorphic to g', and g'' can be generated from h, i.e. $h \Rightarrow_{succs} g''$, where $succs = successorsList_{param}$. Both graphs g and h are partial plane graphs, hence generated from some seed graphs s_g and s_h, respectively.



We can express this property as follows:

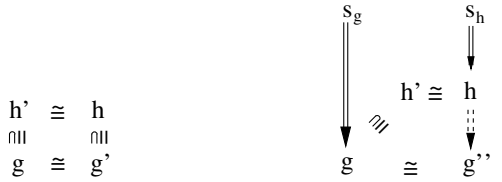
Theorem $s_g \Rightarrow_{succs} g \Longrightarrow s_h \Rightarrow_{succs} h \Longrightarrow set\ (finals\ h) \subseteq set\ (finals\ g') \Longrightarrow g \cong g' \Longrightarrow \exists\, g''.\ g \cong g'' \wedge h \Rightarrow_{succs} g''$!

For some proofs, e.g. the proof of the correctness of our alternative definition of plane graphs, it is sufficient to prove a weaker theorem for the case that g and h are generated from the same initial graph s.



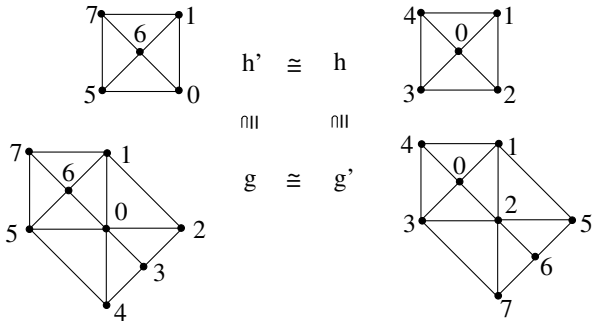
Theorem $s \Rightarrow_{succs} g \Longrightarrow s \Rightarrow_{succs} h \Longrightarrow set\ (finals\ h) \subseteq set\ (finals\ g') \Longrightarrow$
$g \cong g' \Longrightarrow \exists\, g''.\ g' \cong g'' \wedge h \Rightarrow_{succs} g''$ *!*

The conditions $set\ (finals\ h) \subseteq set\ (finals\ g')$ and $g \cong g'$ could also be replaced by $h \cong h'$ and $set\ (finals\ h') \subseteq set\ (finals\ g)$. Both statements are equivalent, which is expressed by the following statement.



Lemma $(\exists\, g'.\ set\ (finals\ h) \subseteq set\ (finals\ g') \wedge g \cong g')$
$= (\exists\, h'.\ h \cong h' \wedge set\ (finals\ h') \subseteq set\ (finals\ g))$ *!*

Example
The graph h is a subgraph of g modulo isomorphism.



There are two different approaches to prove the above theorem: the first approach is by induction on the construction, the second is by induction on the set difference $g - h$.

The first approach requires that we start with the same seed graph. Hence as first step we need to prove that we can start with the same seed graph.

Since the final faces of the seed graph s will be contained in any of the graphs generated from s, it is sufficient to show that we can start the construction with any of the final faces. Thus the proof consists of the following two steps.

1. Prove that for a plane graph g (generated by some seed s_g) we can reach g (up to isomorphism) starting the construction with any of its faces. More precisely, for every face $f \in \mathit{faces}\ g$ we construct a seed graph s_f whose final face is isomorphic to f and then we can generate from s_f a graph g', isomorphic to g.

   Lemma
   $f \in \mathit{set}\ (\mathit{faces}\ g) \Longrightarrow \exists\, g'.\ g \cong g' \wedge \mathit{Seed}\ |\mathit{vertices}\ f| \Rightarrow_{\mathit{succs}} g'$ *!*

2. Prove the weaker theorem about reordering of faces: if we can generate a graph h and a graph g, starting from a seed s, where h is a subgraph (modulo isomorphism) of g, then we can also complete g starting from h. The proof is by induction on the construction of g. It essentially shows that whenever at any point in the construction we can add both a face f_1 and a face f_2, and these faces do not interfere with each other, we can do this in any order.

   Lemma
   $s \Rightarrow_{\mathit{succs}} g \Longrightarrow s \Rightarrow_{\mathit{succs}} h \Longrightarrow \mathit{set}\ (\mathit{finals}\ h) \subseteq \mathit{set}\ (\mathit{finals}\ g') \Longrightarrow g \cong g' \Longrightarrow \exists\, g''.\ g' \cong g'' \wedge h \Rightarrow_{\mathit{succs}} g''$ *!*

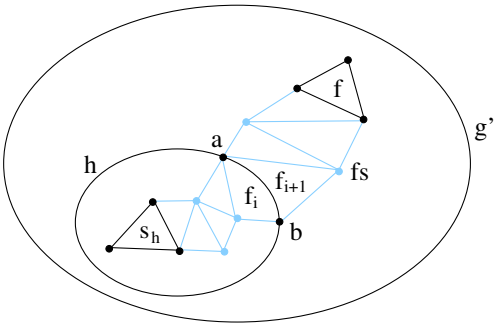
The second part is proved in Isabelle (see Section C.3). The question remains how to formally prove the first part, which is, though easier, a special case of the theorem we set out to prove.

Hence we present another (informal) proof by induction on the set difference of the final faces in g' and in h, using the property that the set of final faces is connected in a plane graph.

start: $\mathit{set}\ (\mathit{finals}\ g') - \mathit{set}(\mathit{finals}\ h) = \{\}$, hence $\mathit{set}\ (\mathit{finals}\ g') = \mathit{set}\ (\mathit{finals}\ h)$, hence $g' = h$. Here we need injectivity of final faces.

step: we obtain $f \in \mathit{set}\ (\mathit{finals}\ g') - \mathit{set}\ (\mathit{finals}\ h)$. Let s_h be the initial face in the construction of h. Both s_h and f are final faces in g. Hence they are connected by a list fs of faces where $hd\ fs = s_h$ and $\mathit{last}\ fs = f$. The head element of fs is contained in $\mathit{set}\ (\mathit{finals}\ h)$ whereas the last element is not. Hence we can obtain an index i where the face $f_i \equiv fs[\![i]\!] \in \mathit{set}\ (\mathit{finals}\ h)$ and $f_{i+1} \equiv fs[\![i{+}1]\!] \notin \mathit{set}\ (\mathit{finals}\ h)$. Both faces

f_i and f_{i+1} lie on two sides of an edge $\{a, b\}$, say $(a, b) \in set\ (edges\ f_i)$ and $(b, a) \in set\ (edges\ f_{i+1})$. Now, since $f_i \in set\ (faces\ h)$, we define f_i' as the unique face in $set\ (faces\ h)$ where $(b, a) \in set\ (edges\ f_i')$. Intuitively, f_i' can be seen as an outer nonfinal face, the boundary of h. The face f_i' is a nonfinal face in h: if $f_i' \in set\ (finals\ h)$ then also $f_i' \in set\ (finals\ g)$ and $f_i' = f_{i+1}$ since both contain the edge (b, a) in g. But f_{i+1} is not in $set\ (finals\ h)$. Contradiction.



Now it remains to show that it is possible to construct a graph h' from h by adding a face isomorphic to f_{i+1} in the nonfinal face $f_i' \in set\ (faces\ h)$ at the edge $(b, a) \in set\ (edges\ f_i')$.

We obtain a graph g''' by renaming g' such that $set\ (finals\ h') \subseteq set\ (finals\ g''')$. Then by induction hypothesis, $set\ (finals\ h')$ can be extended to a graph $g'' \cong g'$, which completes the proof.

We derive the induction rule used in this proof from the rule for induction on the cardinality of a finite set: $(\bigwedge x.\ \forall y.\ card\ (g - y) < card\ (g - x) \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$.

Lemma *finite-subset-rev-induct*: $finite\ g \Longrightarrow P\ g \Longrightarrow$
$(\bigwedge h'.\ h' \subset g \Longrightarrow \exists f.\ f \in g \wedge f \notin h' \wedge (P\ (\{f\} \cup h') \longrightarrow P\ h')) \Longrightarrow$
$h \subseteq g \Longrightarrow P\ h$ $\surd$

A formal proof in Isabelle/HOL would require a formalization of the property of a graph to be *connected*, i.e. for any two faces in a graph there exists a path of faces, connecting both. Then it can be proved by induction on the construction that all plane graphs are connected.

6.2 Fixing a Face and a Edge

In Section 5.1 we introduced an alternative definition of plane graphs which does not compute the successors for all edges in all nonfinal faces, but only for one fixed edge in one fixed nonfinal face. For the completeness theorem we need to show that for every plane graph there is an isomorphic one generated by the optimized definition. We argue informally that whenever we can add two different final faces in a nonfinal graph, and both operations lead to the same final graph, then we can add the faces in any order. Hence the set of graphs is the complete set of plane graphs modulo isomorphism. This definition can also be seen as an alternative definition for the set of plane graphs.

However, in the formal correctness proof of the enumeration algorithm, using the alternative definition does not relieve us from proving the correctness of this refinement step, since the complete formal proof is based on the general definition.

Theoretically, this refinement step could be proved by execution, i.e. by executing a modified enumeration algorithm, where all possible successors graphs are calculated and comparing the result with the archive. Of course, this would immensely increase the number of duplicately calculated isomorphic graphs. A formal proof of the completeness theorem, based on the theorem about reordering of faces, is future work.

Theorem *planeN-complete*:
$g \in \mathit{PlaneGraphs} \Longrightarrow g \in_{\cong} \mathit{PlaneGraphsN}$ *!*

6.3 Restriction of Maximum Face Size to 8

In Section 5.2 we introduced a restriction to the set of plane graphs to graphs with maximum face size 8 (imposed by property tame_1). For the completeness theorem, we need to show that we do not exclude any tame graphs by this restriction. We show that any graph generated by a seed graph with a final face of size greater than 8 is not tame. Since final faces are never deleted during the generation process, every graph contains the final face of the seed graph it was generated from. We call this face the *seed face*. Every graph generated by a parameter greater than 8 contains a seed face of size greater than 8, but property tame_1 does not allow face sizes greater than 8. Subsequently, we present a proof sketch in Isabelle. Consider an

arbitrary graph $g \in PlaneGraphs$ generated by a parameter *param* of type *planeparameter* ($\{3, \ldots\}$). We need to construct a new parameter $param_2$ of type *finparameter* ($\{3, \ldots, 8\}$) and show that g is also generated by this new parameter. This is always possible since the parameter (i.e. the size of the seed face) must be smaller than 8.

Theorem *plane*$_2$*-complete*:
$g \in PlaneGraphsN \Longrightarrow tame\ g \Longrightarrow g \in PlaneGraphs_2$
proof −
assume $g \in PlaneGraphsN$
then obtain *param* **where** $g \in PlaneGraphsN_{param}$
then have $g \in Tree\ successorsList_{param}'\ Seed_{param}$
have *final g*

assume *tame g* **then have** *bound*: $\forall f \in set\ (faces\ g).\ |vertices\ f| \leq 8$

obtain n **where** $param = ptoParam\ n\ n \in planeparameter$
then have $n = toNat_{param}$
have $n \leq 8$
proof
assume $8 < n$
then have $\exists f \in set\ (faces\ g).\ 8 < |vertices\ f|$
with *bound* **show** *False* **by** *auto*
qed

then have $n \in finparameter$
def $param_2 \equiv FtoParam\ n$

have $g \in Tree\ (successorsList\ param_2)\ (Seed\ param_2)$
then have $g \in PlaneGraphs_2\ param_2$
then show *?thesis*
qed

6.4 Complex Seed Graphs

In Section 5.3 we introduced a refinement replacing the first two seed graphs, the triangle and the quadrilateral by a finite set of complex seed graphs. This refinement was divided in 4 steps:

Partitioning of Quad Parameters

The first step of this refinement only separates the parameter set in two distinct sets, hence the set of generated graphs is unchanged. For every graph $g \in \mathit{PlaneGraphs}_2$, generated by a parameter *param* of type *finparameter*, we construct a new parameter *param3* of type *qparameter* or *exceptionalparameter* depending whether *param* is in $\{3, 4\}$ or $\{5, \ldots, 8\}$.

Theorem plane_3*-complete*:
$g \in \mathit{PlaneGraphs}_2 \Longrightarrow g \in \mathit{PlaneGraphs}_3$
proof −
assume $g \in \mathit{PlaneGraphs}_2$
then obtain *param* **where** $g \in \mathit{PlaneGraphs}_{2\mathit{param}}$
then have $g \in \mathit{Tree}\ \mathit{successorsList}_{\mathit{param}}\ \mathit{Seed}_{\mathit{param}}$
have *final g*

obtain *n* **where** $\mathit{param} = \mathit{FtoParam}\ n$ **and** $n \in \mathit{finparameter}$
then have $n = \mathit{toNat}_{\mathit{param}}$

have $n \in \mathit{qparameter} \vee n \in \mathit{exceptionalparameter}$
then show *?thesis*
proof
assume $n \in \mathit{qparameter}$
def $\mathit{param3} \equiv \mathit{Qparameter}\ (\mathit{qtoParam}\ n)$
then have $g \in \mathit{Tree}\ (\mathit{successorsList}_{\mathit{param3}})\ (\mathit{Seed}_{\mathit{param3}})$
have $g \in \mathit{PlaneGraphs}_{3\ \mathit{param3}}$
then show *?thesis*
next
assume $n \in \mathit{exceptionalparameter}$
def $\mathit{param3} \equiv \mathit{Eparameter}\ (\mathit{Eparam}\ n)$
then have $g \in \mathit{Tree}\ (\mathit{successorsList}_{\mathit{param3}})\ (\mathit{Seed}_{\mathit{param3}})$
have $g \in \mathit{PlaneGraphs}_{3\ \mathit{param3}}$
then show *?thesis*
qed
qed

Complex Seed Graphs

For the second step we must show that every plane graph $g \in \mathit{PlaneGraphs}_2$ generated by a triangle or quadrilateral seed (given by a parameter $g \in \mathit{param}$ of type *qparameter*) can be generated by one of the complex seeds (given by

a parameter of type *vertexparameter*).

For a seed of length greater than 4 (given by a parameter *param* of type *exceptionalparameter*), there is nothing to prove, we only need to convert the parameter type.

For a seed of length at most 4, we construct a vertex seed graph. Select an arbitrary vertex v in the seed face of a final graph g. We chose $v{=}0$. The degree of v is at least 3 ($tame_1$). Consider all faces in g that are incident with v. We construct a vertex seed parameter from the cyclic ordered list of the sizes of this faces. From this parameter we construct a vertex seed graph s whose set of final faces is isomorphic to the set of faces of g incident with v.

Using the lemma about reordering of the addition of final faces, we can also reach g from s. Note that for this proof we do not need the assumption that g is tame.

Theorem *plane*$_4$*-complete*:
  $g \in PlaneGraphs_3 \Longrightarrow g \in_{\cong} PlaneGraphs_4$
proof −
 assume $g \in PlaneGraphs_3$
 then obtain *param* **where** $g \in PlaneGraphs_{3\ param}$
 then have $g \in Tree\ successorsList_{param}\ Seed_{param}$
 have *final g*

 show *?thesis*
 proof (*cases param*)
  case (*Qparameter q*)
  then obtain *n* **where** *param* = *Qparameter* (*qtoParam n*)
  and $n \in qparameter$
  then have $n = toNat_{param}$
  then have $n \in \{3,4\}$

  def $v \equiv 0::vertex$
  def $fs \equiv facesAt\ g\ v$
  def $ns \equiv [\ |vertices\ f|\ .\ f \in fs]$
  def $param4 \equiv QParameter\ (vtoParam\ (map\ qtoParam\ ns))$

  then obtain g' **where** $g \cong g'$
  and $g' \in Tree\ successorsList_{param4}\ Seed_{param4}$
  then have $g' \in PlaneGraphs_{4param4}$
  then show *?thesis*
 next

```
  case (Eparameter e)
   then obtain n where param = Eparameter (Eparam n)
   and n ∈ exceptionalparameter
   then have n: n = toNat_param

   def param4 ≡ EParameter (Eparam n)
   then have g ∈ Tree successorsList_param4 Seed_param4
   have g ∈ PlaneGraphs_4param4
   then show ?thesis
  qed
qed
```

Restriction to Tame Complex Seed Graphs

As third step we filter out all vertex seed graphs s with final vertex v where $14.8 \leq \mathbf{b}(tri\ s\ v,\ quad\ s\ v)$.

The correctness of this step is based on the following property: once a vertex v is final in a graph g, it has the same type in all graphs generated from g. Every final graph g' generated from such a seed graph would also contain the vertex v of type $(tri\ s\ v,\ quad\ s\ v)$. Hence the lower bound of the total weight of g' would exceed the target and g' would not be tame. From the definition of **b** (see Section 4.2) we obtain a list of possible types. Up to isomorphism there remain only 17 different vertex seed graphs (see Figure 5.2). We need to show that every seed graph that generates tame graphs is isomorphic to one of the graphs in the list.

```
Theorem plane_5-complete:
    g ∈ PlaneGraphs_4 ⟹ tame g ⟹ g ∈≅ PlaneGraphs_5
proof −
  assume g ∈ PlaneGraphs_4
  then obtain param where g ∈ PlaneGraphs_4param
  then have g ∈ Tree successorsList_param Seed_param
  have final g
  assume tame g

  show g ∈≅ PlaneGraphs_5
  proof (cases param)
   case (QParameter q)
   then obtain n where param = QParameter (vtoParam n)
   and n ∈ vertexparameter
```

then obtain *ns* **where** $n = (map\ qtoParam\ ns)$

def $t \equiv |[i \in ns.\ i = 3]|$
def $q \equiv |[i \in ns.\ i = 4]|$
have b $t\ q\ < 14800$
then have $(t,q) \in \{(2,0), (1,1), (0,2),$
$(3,0), (2,1), (1,2), (0,3),$
$(4,0), (3,1), (2,2), (1,3), (0,4),$
$(5,0), (4,1), (3,2), (2,3), (1,4), (0,5),$
$(6,0), (5,1), (4,2), (3,3), (2,4), (1,5), (0,6)\ \}$
then obtain $param5$::*parameter* **where** $Seed_{param} \cong Seed_{param5}$
then obtain g' **where** $g \cong g'$
and $g' \in Tree\ successorsList_{param5}\ Seed_{param5}$
then have $g' \in PlaneGraphs_{5param5}$
then show *?thesis*
next
case $(EParameter\ e)$
then obtain n **where** $param = EParameter\ (Eparam\ n)$
and $n \in exceptionalparameter$
then have n: $n = toNat_{param}$
def $param5 \equiv ExceptionalParameter\ (Eparam\ n)$
then have $g \in Tree\ successorsList_{param5}\ Seed_{param5}$
have $g \in PlaneGraphs_{5param5}$
then show *?thesis*
qed
qed

Graphs Containing Earlier Seed Graphs

Finally, we neglect graphs that contain an earlier seed graph. Hence we need to show that there is another path in an earlier tree which leads to an isomorphic graph (see Figure 5.3).

Assume that a graph g is generated by a seed s and contains an earlier seed s'. We can obtain a graph g' by renaming the vertices in g such that the final faces of s' are a subset of the final faces in g'. Using the lemma about reordering of faces, we can conclude that there is a graph g'' isomorphic to g' (and also to g) and generated by s'.

The completeness can also be proved by removing this optimization from the algorithm (i.e. replacing the constant "$\mathbf{b'}_{param}$" by "**b**" in the calculation

of a lower bound). Executing the unoptimized algorithm yields two new graphs modulo isomorphism. For these graphs can be verified that they are not tame: they contain a 4-cycle which does not surround one of the tame configurations for 4-cycles.

Moreover, the introduction of complex seed graphs is another optimization of the enumeration algorithm, in the sense that it removes isomorphic duplicates but does not reduce the set of generated graphs modulo isomorphism. Although some seed graphs are excluded which are not tame, these graphs would have also been excluded by the calculation of a lower bound of the total weight. Hence the completeness of this refinement is provable by executing a modified algorithm, starting the generation with a single-faced triangle or quadrilateral graph.

Theorem *plane*$_6$*-complete*:
$g \in \mathit{PlaneGraphs}_5 \Longrightarrow \mathit{tame}\ g \Longrightarrow g \in_{\cong} \mathit{PlaneGraphs}_6$ *!*

6.5 Neglectable by Base Point Symmetry

In Section 5.4 we introduced another refinement step that neglects graphs for which an isomorphic graph is generated by an earlier path in the tree.

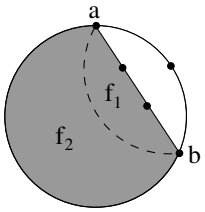
This refinement step is again an optimization in the sense that it removes isomorphic duplicates. We prove the completeness by execution. We execute an unoptimized version of the enumeration algorithm $\mathit{Enumeration}_{\mathit{unopt}}$, obtained by removing this optimization. By construction the optimized version *Enumeration* of the algorithm does not produce more tame graphs than $\mathit{Enumeration}_{\mathit{unopt}}$. We verify that $\mathit{Enumeration}_{\mathit{unopt}}$ does not generate more graphs than *Enumeration*, hence we do not loose tame graphs due to this optimization. This proves the completeness of this optimization. For the completeness proof we might as well remove this optimization from the definition which would cause a slow-down of the enumeration algorithm, but simplify the verification.

Theorem *plane* γ*-complete*:
$g \in \mathit{PlaneGraphs}_6 \Longrightarrow\ g \in_{\cong} \mathit{PlaneGraphs}_\gamma$ $\surd$

6.6 Avoiding Vertices Enclosed by 3-Cycles

In Section 5.5 we introduced a restriction of the set of generated plane graphs using property tame_2. A modification g' of a graph g is neglected when a

3-cycle $[a, b, c]$ would be created which does not surround a single triangle face (see Figure 5.4). To prove the completeness, we show that the arising 3-cycle is neither a face in g nor the opposite of a face in g.



A 3-cycle is created if in a nonfinal face f of a graph g a new edge (a, b) is created, joining two vertices a and b of f which have distance of at least 2 in f and have a common neighbor c in g. Hence a and b have no common neighbor in f and c is not a vertex of f.

By this modification two (final or nonfinal) faces f_1 and f_2 are created such that f_1 contains the edge (b, a) and f_2 contains the edge (a, b). Both faces f_1 and f_2 contain only vertices of f, hence c is neither a vertex of f_1 nor f_2 and both f_1 and f_2 are not equivalent to $[a, b, c]$.

Using the property of plane graphs, that there is exactly one face in g that contains an edge (a, b) of g, there exists no face in g which equivalent to $[a, b, c]$. Similarly there exists no face in g which equivalent to $[c, b, a]$. The formal proof is future work.

6.7 Lower Bounds for the Total Weight

First we show the correctness of the lower bound for the total weight for the case of a final graph. This an important step in the proof, since the calculation of the lower bound for nonfinal graphs is based on this lower bound.

Theorem
For a final graph g, generated by any parameter p, $squanderLowerBound_{param}\ g$ is a lower bound for the total weight $\sum_{f \in faces\ g} w(f)$ for all admissible weight assignments w.

Proof
Let $F = finals\ g = faces\ g$.
By definition, we have

(S) $faceSquanderLowerBound_{param}\ g = \sum_{f \in F} \mathbf{d}\ |vertices\ f|$.

By definition we obtain a list of vertices V where

(N) $ExcessNotAt_p\ g\ None = \sum_{v \in V} ExcessAt_p\ g\ v$.

We partition V in V_1 and V_2 and V_2 in V_3 and V_4:

$V_1 = [v \in V\ .\ except\ g\ v = 0]$
$V_2 = [v \in V\ .\ except\ g\ v \neq 0]$
$V_3 = [v \in V\ .\ except\ g\ v \neq 0 \wedge degree\ g\ v = 5]$
$V_4 = [v \in V\ .\ except\ g\ v \neq 0 \wedge degree\ g\ v \neq 5]$

We partition the set of faces F in F_1 and F_2 and F_2 in F_3 and F_4:

$F_1 = [f \in F\ .\ set\ (vertices\ f) \cap set\ V_1 \neq \{\}]$
$F_2 = [f \in F\ .\ set\ (vertices\ f) \cap set\ V_1 = \{\}]$
$F_3 = [f \in F_2\ .\ set\ (vertices\ f) \cap set\ V_3 \neq \{\}]$
$F_4 = [f \in F_2\ .\ set\ (vertices\ f) \cap set\ V_3 = \{\}]$

With the definition of $ExcessAt_p\ g\ v$ we have

(E_1) $\sum_{v \in V_1} ExcessAt_p\ g\ v + \sum_{f \in F_1} \mathbf{d}\ |vertices\ f| = \sum_{v \in V_1} \mathbf{b'}_p\ (tri\ g\ v)\ (quad\ g\ v)$

(E_2) If $v \in V_3$ then $ExcessAt_p\ g\ v = \mathbf{a}\ (tri\ g\ v)$.

(E_3) If $v \in V_4$ then $ExcessAt_p\ g\ v = 0$.

From admissibility of w we have

(A_1) $\sum_{v \in V_1} \mathbf{b'}_p\ (tri\ g\ v)\ (quad\ g\ v) \leq \sum_{f \in F_1} w(f)$ $\quad (admissible_2)$

(A_2) $\sum_{v \in V_3} \mathbf{a}\ (tri\ g\ v) + \sum_{f \in F_3} \mathbf{d}\ |vertices\ f| \leq \sum_{f \in F_3} w(f)$ $\quad (admissible_4)$

(A_3) $\sum_{f \in F_4} \mathbf{d}\ |vertices\ f| \leq \sum_{f \in F_4} w(f)$ $\quad (admissible_4)$

Finally, we calculate

$$
\begin{array}{lcl}
 & & \mathit{squanderLowerBound}_p\ g \\
(\text{Def.}) & = & \mathit{ExcessNotAt}_p\ g\ \mathit{None} + \mathit{faceSquanderLowerBound}_p\ g \\
(\text{N,S}) & = & \sum_{v \in V} \mathit{ExcessAt}_p\ g\ v + \sum_{f \in F} \mathbf{d}\ |\mathit{vertices}\ f| \\
 & = & \sum_{v \in V_1} \mathit{ExcessAt}_p\ g\ v + \sum_{v \in V_2} \mathit{ExcessAt}_p\ g\ v \\
 & & + \sum_{f \in F_1} \mathbf{d}\ |\mathit{vertices}\ f| + \sum_{f \in F_2} \mathbf{d}\ |\mathit{vertices}\ f| \\
(E_1) & = & \sum_{v \in V_1} \mathbf{b'}_p\ (\mathit{tri}\ g\ v)\ (\mathit{quad}\ g\ v) \\
 & & + \sum_{v \in V_2} \mathit{ExcessAt}_p\ g\ v + \sum_{f \in F_2} \mathbf{d}\ |\mathit{vertices}\ f| \\
 & = & \sum_{v \in V_1} \mathbf{b'}_p\ (\mathit{tri}\ g\ v)\ (\mathit{quad}\ g\ v) + \sum_{v \in V_3} \mathit{ExcessAt}_p\ g\ v \\
 & & + \sum_{v \in V_4} \mathit{ExcessAt}_p\ g\ v + \sum_{f \in F_2} \mathbf{d}\ |\mathit{vertices}\ f| \\
(E_2,\ E_3) & = & \sum_{v \in V_1} \mathbf{b'}_p\ (\mathit{tri}\ g\ v)\ (\mathit{quad}\ g\ v) + 0 + \sum_{v \in V_3} \mathbf{a}\ (\mathit{tri}\ g\ v) \\
 & + & \sum_{f \in F_3} \mathbf{d}\ |\mathit{vertices}\ f| + \sum_{f \in F_4} \mathbf{d}\ |\mathit{vertices}\ f| \\
(A_1,\ A_2,\ A_3) & \leq & \sum_{f \in F_1} w(f) + \sum_{f \in F_3} w(f) + \sum_{f \in F_4} w(f) \\
 & = & \sum_{f \in F} w(f)
\end{array}
$$

Qed

The formal Isabelle/Isar proof can be found in Appendix C.

Theorem *total-weight-lowerbound*:
$g \in \mathit{PlaneGraphs}_{8\ \mathit{param}} \Longrightarrow$
$\mathit{admissible}\ w\ g \Longrightarrow \sum_{f\ \in\ \mathit{faces}\ g} w\ f < \mathit{squanderTarget} \Longrightarrow$
$\mathit{squanderLowerBound}_{\mathit{param}}\ g\ \leq \sum_{f\ \in\ \mathit{faces}\ g} w\ f$ $\surd$

The lower bound of a nonfinal graph is based on this lower bound. The formal proof is future work.

6.8 Nonfinal Triangles and Quadrilaterals

For the correctness of the functions for handling of nonfinal triangles *makeTrianglesFinal* and quadrilaterals *handleQuad* (see Section 5.8) we need to verify them against the definition of properties $tame_2$ and $tame_3$.

The function *makeTrianglesFinal* replaces all nonfinal triangles by final ones, since every 3-cycle in a tame graph is either a triangle or the opposite of a triangle (see Section 5.8).

The function *handleQuad* replaces every nonfinal quadrilateral by all possible tame configurations bounded by a 4-cycle (see Section 5.8). The function *handleQuad* is based on the functions *FaceDivisionGraph* and *makeFaceFinal* which split a face or make a nonfinal face final, respectively. On the other hand the definition of plane graphs uses the function *addFace*, which calls *FaceDivisionGraph* and *makeFaceFinal* on certain arguments. But *handleQuad* is based also on *FaceDivisionGraph* and *makeFaceFinal*. Therefore it must be shown that every completion of a nonfinal quadrilateral with one of the tame configurations can also be constructed by a sequence of *addFace* operations with appropriate arguments. Moreover it must be verified that all other successors are not tame. This proof is future work.

6.9 Neglectable Nonfinal Graphs

In section Section 5.9 we introduced some sufficient conditions that a nonfinal graphs can be neglected. The correctness of this refinement step is based on the correctness of the lower bounds for the total weight and the correctness of avoiding vertices enclosed by 3-cycles. The formal proof is future work.

Theorem *plane*$_8$*-complete*:
$g \in PlaneGraphs_7 \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} PlaneGraphs_8$ *!*

6.10 Neglectable Final Graphs

In Section 5.9 we introduced some conditions for neglectable final graphs. Subsequently, we show the completeness of this last refinement step of the enumeration.

Theorem *Enumeration-complete*:
$g \in PlaneGraphs_8 \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} Enumeration$ $\surd$

A graph is neglected if it contains a vertex of type $(1, 0, 1)$. First we prove informally that no vertices of type $(1, 0, 1)$ are contained in a tame graph. A vertex of type $(1, 0, 1)$ is incident with exactly one triangle, and exactly one face of size 5.

Finally, we formally prove that whenever a final graph is neglected, it can not be tame.

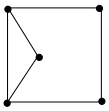


Figure 6.1: A vertex of type $(1, 0, 1)$

Lemma
In a tame plane graph, there cannot be a vertex of type $(1, 0, 1)$.

Proof
We can show that every graph that contains a vertex of this type is not tame. This configuration is bounded by a 4-cycle, hence on the outside there can only be one of the tame configurations ($tame_3$), but none of them leads to a graph with at least 8 triangles, and $\sum_{f \in faces\ g} \mathbf{c}\ |vertices\ f| < 4$ which contradicts property $tame_6$.
Qed

The formal proof remains future work.

Lemma *PlaneGraphs*$_8$*-not-has101Type*:
$g \in PlaneGraphs_8 \Longrightarrow tame\ g \Longrightarrow \neg\ has101Type\ g$ *!*

Finally, we show that a graph g can be neglected if

- it contains a vertex of degree greater than 6 ($tame_4$),
- it contains an exceptional vertex of degree greater than 5 ($tame_5$),

- *scoreUpperBound* g < *8000*, since *scoreUpperBound* is a lower bound for $\sum_{f \in \mathit{faces}\ g}$ **c** |*vertices* f| and *8000* $\leq \sum_{f \in \mathit{faces}\ g}$ **c** |*vertices* f| (*tame*$_6$),
- *squanderTarget* $\leq$ *ExcessNotAt*$_{\mathit{param}}$ g *None* + *faceSquanderLowerBound*$_{\mathit{param}}$ g, since this is a lower bound for $\sum_{f \in \mathit{faces}\ g}$ $w\ f$ for all admissible weight assignments with $\sum_{f \in \mathit{faces}\ g}$ $w\ f$ < *squanderTarget* and $\exists\, w.$ *admissible* $w\ g \wedge \sum_{f \in \mathit{faces}\ g}$ $w\ f$ < *squanderTarget* (*tame*$_7$),
- g contains two adjacent vertices of type $(4, 0)$ (*tame*$_8$), and
- g contains a vertex with one adjacent triangle and one adjacent pentagon.

We formally prove that no tame graphs are neglectable final graphs.

Lemma *PlaneGraphs*$_8$*-not-neglectableFinal*:
$g \in \mathit{PlaneGraphs}_{8\ \mathit{param}} \Longrightarrow \mathit{tame}\ g \Longrightarrow \neg\ \mathit{neglectableFinal}_{\mathit{param}}\ g$
proof −
assume $g \in \mathit{PlaneGraphs}_{8\ \mathit{param}}$
then obtain $g \in \mathit{PlaneGraphs}_8$
then obtain *final* g

assume *tame* g

show $\neg\ \mathit{neglectableFinal}_{\mathit{param}}\ g$
proof
assume $\mathit{neglectableFinal}_{\mathit{param}}\ g$
then show *False*
proof (*cases*)
fix v **assume** $v \in \mathit{set}\ (\mathit{vertices}\ g)$ **and** *except* $g\ v = 0$ **and** $6 < \mathit{degree}\ g\ v$
then show *False* — (*tame*$_4$)
next
fix v **assume** $v \in \mathit{set}\ (\mathit{vertices}\ g)$ **and** $0 < \mathit{except}\ g\ v$ **and** $5 < \mathit{degree}\ g\ v$
then show *False* — (*tame*$_5$)
next
obtain w **where** *admissible* $w\ g$
and $\sum_{f \in \mathit{faces}\ g}$ $w\ f$ < *squanderTarget* — (*tame*$_7$)
assume *squanderTarget* $\leq$ *squanderLowerBound*$_{\mathit{param}}$ g
also have ... $\leq$ $\sum_{f \in \mathit{faces}\ g}$ $w\ f$ **by** (*rule total-weight-lowerbound*)
finally have $\neg$ $\sum_{f \in \mathit{faces}\ g}$ $w\ f$ < *squanderTarget* **by** *auto*
then show *False* **by** *contradiction*
next
assume *scoreUpperBound* g < *scoreTarget*

```
    then show False  — ( tame 6 )
   next
    assume  hasAdjacent40 g
    then show False — ( tame 8 )
   next
    assume has101Type g
    then show False                                                    !
   qed
  qed
qed
```

We show that the set of generated graphs is exactly the set of graphs of generated by $PlaneGraphs_8$ that are not neglectable final graphs. The proof of this equality can be found in Appendix C.

Lemma *PlaneGraphs*$_8$*Param-eq*: *EnumerationParam param*
$= \{g.\ g \in PlaneGraphs_{8\ param} \wedge \neg\ neglectableFinal_{param}\ g\ \}$

We conclude the completeness theorem of the enumeration.

Theorem *Enumeration-complete*:
$g \in PlaneGraphs_8 \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} Enumeration$
proof −
assume *tame*: *tame g*
assume *g*: $g \in PlaneGraphs_8$
then obtain *param* **where** *p*: $g \in PlaneGraphs_{8\ param}$ **by** *auto*
have $\neg\ neglectableFinal_{param}\ g$ **by** (*rule PlaneGraphs*$_8$*-not-neglectableFinal*)
with *p* **have** $g \in EnumerationParam\ param$ **by** (*simp add*: *PlaneGraphs*$_8$*Param-eq*)

then have $g \in Enumeration$ **by** *auto*
then show *?thesis*
qed

6.11 Summary

Now, we summarize all completeness theorems of Chapter 6 and finally prove the completeness of the enumeration:

By construction, $Enumeration \subseteq_{\cong} PlaneGraphs_8 \subseteq_{\cong} PlaneGraphs_7 \subseteq_{\cong} .. \subseteq_{\cong} PlaneGraphs$.

The correctness of the alternative definition of plane graphs is informally proved.

Theorem *planeN-complete*:
$g \in PlaneGraphs \Longrightarrow g \in_{\cong} PlaneGraphsN$

The restriction of face size to 8 is formally proved.

Theorem *plane$_2$-complete*:
$g \in PlaneGraphsN \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} PlaneGraphs_2$ $\surd$

The introduction of complex seed graphs is informally proved. Future work is the proof by execution or using the theorem about reordering of faces.

Theorem *plane$_3$-complete*:
$g \in PlaneGraphs_2 \Longrightarrow g \in PlaneGraphs_3$ $\surd$

Theorem *plane$_4$-complete*:
$g \in PlaneGraphs_3 \Longrightarrow g \in_{\cong} PlaneGraphs_4$ *!*

Theorem *plane$_5$-complete*:
$g \in PlaneGraphs_4 \Longrightarrow tame\ g \Longrightarrow\ g \in_{\cong} PlaneGraphs_5$ *!*

Theorem *plane$_6$-complete*:
$g \in PlaneGraphs_5 \Longrightarrow tame\ g \Longrightarrow\ g \in_{\cong} PlaneGraphs_6$ *!*

The optimization of graphs neglectable by base point symmetry is proved by execution.

Theorem *plane$_7$-complete*:
$g \in PlaneGraphs_6 \Longrightarrow\ g \in_{\cong} PlaneGraphs_7$ $\surd$

The proof of correctness neglecting nonfinal graphs is future work, based on the correctness of avoiding of 3-cycles with enclosed vertices and of the lower bounds for nonfinal graphs.

Theorem *plane$_8$-complete*:
$g \in PlaneGraphs_7 \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} PlaneGraphs_8$ *!*

The correctness of the lower bound for final graphs is formally proved. The correctness of neglecting final graphs is formally proved, except for the correctness of excluding graphs that contain vertices of type $(1, 0, 1)$ which is informally proved.

Theorem *Enumeration-complete*:
$g \in PlaneGraphs_8 \Longrightarrow tame\ g \Longrightarrow g \in_{\cong} Enumeration$ *(!)*

We generate ML code from the definition of *Enumeration*, executing it and check that for every generated graph there is an isomorphic graph in the archive. Thereby we obtain the following result;

Theorem *Archive-complete*:
$g \in \mathit{Enumeration} \Longrightarrow g \in_{\cong} \mathit{Archive}$ $\surd$

This finally proves the correctness of Hales' algorithm for the enumeration of all tame plane graphs:

Theorem $g \in \mathit{PlaneGraphs} \Longrightarrow \mathit{tame}\ g \Longrightarrow g \in_{\cong} \mathit{Archive}$

Chapter 7

Conclusion

The proof of the Kepler conjecture presented by Thomas C. Hales is a proof by exhaustion on a finite set of plane graphs with certain properties, called tame plane graphs. This set of tame plane graphs, called the *Archive*, is generated by a Java program. Hence an essential part of the proof is to show completeness of this enumeration.

We contributed on the formalization of the completeness of this enumeration algorithm.

7.1 Summary

Formalization of Plane Graphs We developed a theory of plane graphs in Isabelle/HOL, based on an inductive definition: a connected plane graph is constructed by starting with one face and repeatedly adding new faces. We validated our definition of plane graphs: we informally proved that we reach all plane graphs by this definition.

We proved correctness theorems of the construction operations of a plane graph and provided an induction principle, which can be used to prove properties of plane graphs by induction on the construction. Moreover, we exemplarily carried out some induction proofs for properties of plane graphs. We formalized plane graph isomorphisms and showed informally that for a given graph the faces can be added in any order, as long as every graph during the construction is connected.

Formalization of an Enumeration Algorithm We defined an executable Isabelle/HOL function *Enumeration*, containing a superset of all tame plane graphs. We generated ML code using Isabelle/HOL's code generator and execute the ML code, comparing the result with the *Archive*. We succeeded to enumerate all graphs of the *Archive*. Hence we obtained a validation of the formalization of *Enumeration*.

We simplified the original algorithm. Firstly, we replaced iterative definitions by more declarative definitions, which are more appropriate for a correctness proof, but still produce the same set of graphs. Moreover, we identified unnecessary optimizations, which reduce the number of generated graphs, but modulo isomorphism generate the same set of graphs. Removing these optimizations from the algorithm further simplifies the completeness proof.

Completeness Proof We formalized a notion of tameness. We uncovered a mismatch of the definition of tame graph in Hales' paper and the Java program provided by Hales. The original definition of tameness included graphs which are not in the *Archive*. This had the consequence that the definition of tame graphs in the proof of the Kepler conjecture needed to be changed (see section 4.5). Fortunately, the correctness Hales' proof of the Kepler conjecture was not affected by this change.

We formalized the completeness of the enumeration:

Theorem
For every tame plane graph there is an isomorphic graph in the *Archive*.

We partitioned the proof into subproofs, identified proof obligations and informally proved all steps. Essential parts were formally proved.

7.2 Approach

We proposed an approach for program verification of imperative programs (in our case a Java program) in a theorem prover:

- Translate the program to a Isabelle/HOL, considering Isabelle/HOL as a functional programming language.
- Prove the correctness of the Isabelle functions. Program verification for functional programs is by far easier than for imperative programs.

- Generate executable ML code from the Isabelle/HOL definition using Isabelle/HOL's code generator.
- Validate the formalization by comparing the results. In the special case of a finite calculation, as in our formalization of Hales' proof, we completely tested our formalization.

7.3 Future Work

Future work is to complete the open points in the formal verification of the enumerations. The main points are:

- The lower bound for the total weigth of an admissible weight assignment, calculated for nonfinal graphs, need to be formally verified. The verification is based on the existing formal proof of a lower bound calculated for final graphs.
- The treatment of nonfinal triangles and quadrilaterals and the exclusion of vertices enclosed by triangles needs to be verified against the Isabelle definition of tame graphs.
- The proof requires invariants of plane graphs that can be proved by induction on the construction of a plane graph.
- It has to be formally verified that the definition of tameness excludes vertices with one incident triangle and one incident pentagon. The proof requires a special case of Jordan's Curve theorem.

Informally, we have argued that all these open points are valid. Hence, though the formal correctness proof has not yet been finished, we already gained confirmation of the completeness of Hales' enumeration of tame plane graphs.

Appendix A

Algorithms

A.1 List Functions

List Intersection

The function $as \cap bs$ calculates the list of all elements x where $x \in set\ as$ and $x \in set\ bs$.

$[] \cap bs = []$
$(a\#as) \cap bs = (if\ a \in set\ bs\ then\ a\#(as \cap bs)\ else\ as \cap bs)$

Lemma $set\ (as \cap bs) = set\ as \cap set\ bs$ $\surd$

Minimal and Maximal Elements of a List

The function $minimal\ m\ a\ bs$ returns the minimal element of the set $\{a\} \cup set\ bs$ with respect to a rating function m.

$minimal :: ('a \Rightarrow nat) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a$
$minimal\ m\ a\ [] = a$
$minimal\ m\ a\ (b\#bs) =$
$\quad (let\ mbs = minimal\ m\ a\ bs\ in$
$\quad if\ m\ b \leq m\ mbs\ then\ b\ else\ mbs)$

Lemma $minimal\ m\ a\ bs \in \{a\} \cup set\ bs$ √

Lemma $\bigwedge x.\ x \in \{a\} \cup set\ bs \Longrightarrow m\ (minimal\ m\ a\ bs) \leq m\ x$ √

The functions *minList a bs* and *maxList a bs* calculate the minimal and maximal element the set $\{a\} \cup set\ bs$

$minList\ a\ [] = a$
$minList\ a\ (b\#bs) = minList\ (min\ a\ b)\ bs$

Lemma $\bigwedge a.\ minList\ a\ bs \in \{a\} \cup set\ bs$ √

Lemma $\bigwedge x\ a.\ x \in \{a\} \cup set\ bs \Longrightarrow minList\ a\ bs \leq x$ √

$maxList\ a\ [] = a$
$maxList\ a\ (b\#bs) = maxList\ (max\ a\ b)\ bs$

Lemma $\bigwedge a.\ maxList\ a\ bs \in \{a\} \cup set\ bs$ √

Lemma $\bigwedge x\ a.\ x \in \{a\} \cup set\ bs \Longrightarrow x \leq maxList\ a\ bs$ √

Replacing Elements in a List

The function *replace ls oldF fs* replaces an element *oldF* of a list *ls* by a list of new elements *fs*.

$replace\ oldF\ newFs\ [] = []$
$replace\ oldF\ newFs\ (l\#ls) =$
$\quad (if\ l = oldF\ then\ newFs\ @\ ls\ else\ l\#(replace\ oldF\ newFs\ ls))$

We summarize the characteristic properties of *replace oldF newFs ls*. Elements of the replaced list are either elements of the original list *ls* or the new elements of the list *newFs*. If *oldF* is not an element of *ls* then the list is unchanged. If *oldF* is an element of *ls* then all elements of *newFs* are in the replaced list. Furthermore the replaced list contains all elements of *ls* different from *oldF* and it does not contain *oldF* provided *oldF* is not one of the new elements.

Lemma $f \in set\ (replace\ oldF\ newFs\ ls) \Longrightarrow f \notin set\ ls \Longrightarrow f \in set\ newFs$ √

Lemma $oldF \notin set\ ls \Longrightarrow replace\ oldF\ newFs\ ls = ls$ $\surd$

Lemma $oldF \in set\ ls \Longrightarrow f \in set\ newFs \Longrightarrow f \in set\ (replace\ oldF\ newFs\ ls)$ $\surd$

Lemma $f \in set\ ls \Longrightarrow oldF \neq f \Longrightarrow f \in set\ (replace\ oldF\ newFs\ ls)$ $\surd$

Lemma $oldF \notin set\ newFs \Longrightarrow distinct\ ls \Longrightarrow$
$oldF \notin set\ (replace\ oldF\ newFs\ ls)$ $\surd$

Applying a Function to a List at a Set of Indices

We introduce an auxiliary function *mapAt* on list. *mapAt ns f as* applies a function f on all elements of positions given by a list of indices *ns*.

$mapAt\ []\ f\ as = as$
$mapAt\ (n\#ns)\ f\ as =$
$\quad (if\ n < |as|\ then\ mapAt\ ns\ f\ (as[n{:=}\ f\ (as[\![n]\!])])$
$\quad else\ mapAt\ ns\ f\ as)$

Removing Elements in a Table

The function *removeKeyList ws ps* removes all entries $(w,\ e)$ from a table *ps* if $w \in set\ ws$.

$('a,\ 'b)\ table = ('a \times 'b)\ list$

$removeKey :: 'a \Rightarrow ('a,\ 'b)\ table \Rightarrow ('a,\ 'b)\ table$
$removeKey\ a\ ps \equiv [p \in ps.\ a \neq fst\ p]$

$removeKeyList :: 'a\ list \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$
$removeKeyList\ []\ ps = ps$
$removeKeyList\ (w\#ws)\ ps = removeKey\ w\ (removeKeyList\ ws\ ps)$

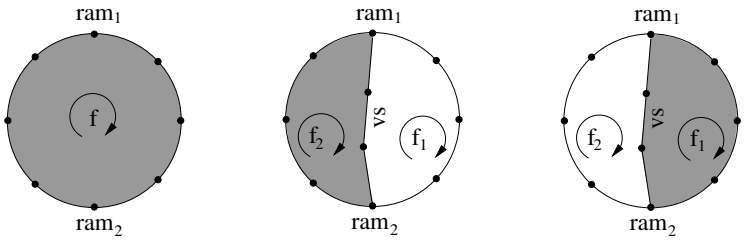


Figure A.1: Comparison of *addFaceInv* (middle) and *addFace* (right).

A.2 Complex Seed Graphs

The function *addFaceInv* is defined similarly as *addFace*, only the actual representation of generated graphs differs in the order the set of faces is represented in the face list and the order vertices are represented in the vertex lists of the faces. We followed the representation in Hales' Java program, because this made it easier to generate an ML program from the Isabelle definition that produced exactly the same set of graphs as the Java program. For simplicity, the function *addFaceInv* could be replaced by the function *addFace* (compare Section 3.2.4). This would change the set of generated graphs only modulo graph isomorphism. Hence the test if the generated set of graphs is contained in the archive (modulo graph isomorphism) still succeeds. The current definition has the only disadvantage that some proofs for *addFace* must be repeated for *addFaceInv*. Of course, both proofs are quite similar.

We summarize the differences between the functions *addFace* and *addFaceInv*: for both functions the vertex list argument *vs* is given in the order in which they occur in face f_2; after each face split the function *addFaceInv* makes the face f_1 final, whereas *addFace* leaves all faces nonfinal, only as last step the face f_2 is made final. The recursion is in both cases on the face f_2.

Example
The Figure A.1 shows the results of the operations *addFaceInv g f* [ram_1, ram_2] (middle) and *addFace g f* [ram_1, ram_2] (right) applied to a nonfinal face *f* (left).

```
addFaceInvSnd :: graph ⇒ face ⇒ vertex ⇒ nat ⇒ vertex option list ⇒ graph
addFaceInvSnd g f w1 n [] = g
addFaceInvSnd g f w1 n (v#vs) =                (* n = number of new vertices *)
  (case v of None ⇒ addFaceInvSnd g f w1 (Suc n) vs
    | (Some w2) ⇒
```

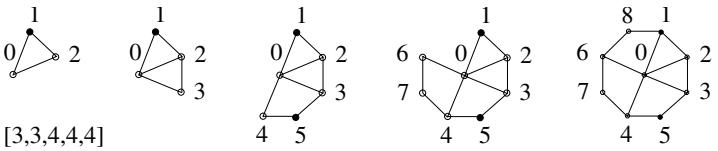


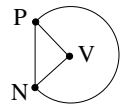
Figure A.2: Construction of a seed face

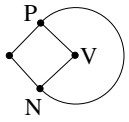
```
        if nextVertex f w_1 = w_2 ∧ n = 0
        then addFaceSnd g f w_2 0 vs
        else let ws = [countVertices g ..< countVertices g + n];
        (f_1, f_2, g') = FaceDivisionGraph g w_1 w_2 f (rev ws);
        g' = makeFaceFinal f_1 g' in
        addFaceInvSnd g' f_2 w_2 0 vs)

addFaceInv :: graph ⇒ face ⇒ vertex option list ⇒ graph
addFaceInv g f [] = g
addFaceInv g f (v#vs) =
    (* search for starting point:
       vertex followed by null or a non−adjacent edge *)

    (case v of None ⇒ addFaceInv g f vs
        | (Some w_1) ⇒ addFaceInvSnd g f w_1 0 vs)
```

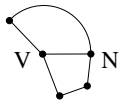
The function *SeedGraph ls* generates a vertex graph with face lengths of the final faces arranged *clockwise* around one final vertex given by *ls* (see Figure A.2). We assume that all elements of *ls* are in the set $\{3, 4\}$ and the length of *ls* is at least 2.

First, a seed graph of the length of *hd ls* is created. In the next step, the first nonfinal face $F = (nonFinals\ g)[\![0]\!]$ of the seed graph is modified: If there is more than one face to add then a face of appropriate size is added inside the nonfinal face, containing the vertex $V = 0$ and the next vertex $N = F{\cdot}V$ of V in *f*. If there is only one face to add, a face of appropriate size is added inside the nonfinal face, containing the previous vertex $P = F^{-1}{\cdot}V$ and the next vertex $N = F{\cdot}V$ of $V = 0$ in *f*.

```
addSeedFaces :: graph ⇒ nat list ⇒ graph
addSeedFaces g [] = g
addSeedFaces g (n#ns) =
   (let F = (nonFinals g)⟦0⟧;
    V = 0; N = F·V; P = F^-1·V in
    case ns of [] ⇒
      if n = 3
      then addFaceInv g F ([Some P, Some N])
      else if n = 4
      then addFaceInv g F [Some P, None, Some N]
      else g
    | (m#ms) ⇒
      if n = 3 then let
      g' = addFaceInv g F [Some V, None, Some N] in
      addSeedFaces g' ns
      else if n = 4 then let
      g' = addFaceInv g F [Some V, None, None, Some N] in
      addSeedFaces g' ns
      else g)

SeedGraph (n#ns) = addSeedFaces (graph n) ns
```







A.3 Conversion of Graph Representations

In the archive provided by Hales the graphs are stored in a certain format, as a list of integers. In order to be able to compare generated graphs with graphs in the archive we need conversion functions between the different representations. We describe the format of the graphs in the archive and provide conversion functions from the graph data structure to the archive format.

Graphs in the archive are encoded as a list of integers, representing a list of faces.

- The head element in the list represents the number of nonfinal faces followed by a (possibly empty) sequence of indices of the nonfinal faces in the face list.
- Then follows the number of faces.

- Then follows the encoding of the faces, stringed together. The encoding of a face starts with the number of vertices, followed by the sequence of vertices.

Example
The following list of integers

$[0,\ 17,\ \ 3,\ 0,\ 1,\ 2,\ \ 3,\ 3,\ 2,\ 9,\ \ 3,\ 3,\ 0,\ 2,\ \ 4,\ 5,\ 4,\ 0,\ 3,\ \ 4,\ 7,\ 6,\ 0,\ 4,$
$4,\ 8,\ 1,\ 0,\ 6,\ \ 3,\ 9,\ 2,\ 10,\ \ 3,\ 10,\ 2,\ 1,\ \ 3,\ 10,\ 1,\ 8,\ \ 3,\ 10,\ 8,\ 11,$
$3,\ 9,\ 10,\ 11,\ \ 3,\ 11,\ 8,\ 7,\ \ 3,\ 8,\ 6,\ 7,\ \ 3,\ 7,\ 4,\ 5,\ \ 3,\ 5,\ 3,\ 9,$
$3,\ 11,\ 7,\ 5,\ \ 3,\ 5,\ 9,\ 11]]$

is an encoding of a final graph with the following set of 17 faces:

$\{\ [0,\ 1,\ 2],\ [3,\ 2,\ 9],\ [3,\ 0,\ 2],\ [5,\ 4,\ 0,\ 3],\ [7,\ 6,\ 0,\ 4],$
$[8,\ 1,\ 0,\ 6],\ [9,\ 2,\ 10],\ [10,\ 2,\ 1],\ [10,\ 1,\ 8],\ [10,\ 8,\ 11],$
$[9,\ 10,\ 11],\ \ [11,\ 8,\ 7],\ [8,\ 6,\ 7],\ [7,\ 4,\ 5],\ [5,\ 3,\ 9],$
$[11,\ 7,\ 5],\ [5,\ 9,\ 11]\ \}.$

Output

The function *toArchiveString* calculates the conversion from the graph datatype representation to the archive format.

```
pos :: 'a list ⇒ 'a ⇒ nat
pos [] a = 0
pos (b#bs) a = (if a = b then 0 else 1 + (pos bs a))

nonFinalsPos :: graph ⇒ nat list
nonFinalsPos g ≡ [pos (faces g) f. f ∈ (nonFinals g)]

toArchiveStringFaceList :: face list ⇒ nat list
toArchiveStringFaceList [] = []
toArchiveStringFaceList (f#fs) =
    (let vs = vertices f in [ |vs| ]@vs@toArchiveStringFaceList fs)

toArchiveString :: graph ⇒ nat list
toArchiveString g ≡
    [ |nonFinals g| ]@(nonFinalsPos g)@
    [ |faces g| ]@(toArchiveStringFaceList (faces g))
```

Input

The function *formatter* calculates the conversion from the archive format to the graph datatype. First the list of faces is constructed, assigning each face the correct face type, then for every vertex v the list of incident faces is created by sorting the incident faces such that they follow each other in counterclockwise order around v. The *baseVertex* and the heights are set to the initial value 0.

The function *formatterFaces* constructs the face list.

```
setNonFinal :: face ⇒ face
setNonFinal f ≡ Face (vertices f) Nonfinal

formatterSetTemps :: nat list ⇒ face list ⇒ face list
formatterSetTemps ns fs ≡ mapAt ns setNonFinal fs

formatterVertexLists  :: nat list ⇒ nat list list
formatterVertexLists [] = []
formatterVertexLists (n#ns) =
    (take n ns)#formatterVertexLists (drop n ns)

formatterFaces :: nat list ⇒ nat list list ⇒ face list
formatterFaces templist fs ≡
    formatterSetTemps templist [Face vs Final. vs ∈ fs]
```

The function *formatterFacesAt* constructs the incidence lists.

```
position-mapRec :: (nat ⇒ 'a ⇒ 'b) ⇒ nat ⇒ 'a list ⇒ 'b list
position-mapRec f n [] = []
position-mapRec f n (x # xs) = f n x # position-mapRec f (Suc n) xs

position-map :: (nat ⇒ 'a ⇒ 'b) ⇒ 'a list ⇒ 'b list
position-map f ≡ position-mapRec f 0
```

For a fixed vertex v, we collect all faces that contain v.

```
insertFaces :: nat ⇒ face list ⇒ face list list
insertFaces n fs ≡ [ [f ∈ fs.  i ∈ set (vertices f)] .i ∈ [0..<n]]
```

Then we sort the face list.

```
findNextFace :: face list ⇒ vertex ⇒ vertex ⇒ face
findNextFace (f#fs) v w =
```

```
    (if w mem (vertices f) ∧ f·w = v then f else findNextFace fs v w)

sortNextFace :: vertex ⇒ vertex ⇒ nat ⇒ face list ⇒ face list
sortNextFace v w 0 fs = []
sortNextFace v w (Suc n) fs =
    (let f = findNextFace fs v w in f # sortNextFace v (f·v) n (rem f fs))

sortFaces :: vertex ⇒ face list ⇒ face list
sortFaces v [] = []
sortFaces v (f#fs) = f # (sortNextFace v (f·v) (length fs) fs)

formatterFacesAt :: nat ⇒ face list ⇒ face list list
formatterFacesAt n  fs ≡
    (position-map (sortFaces) (insertFaces n fs))

formatter :: nat list ⇒ graph formatter [] = emptyGraph
formatter (t#ts) =
    (let templist = take t ts;
    n = hd (drop t ts);
    ns = tl (drop t ts);
    fs = formatterVertexLists ns;
    m = maxList 0 [maxList 0 f. f ∈ fs] + 1;        (* number of vertices *)
    facelist = formatterFaces templist fs in
    Graph facelist
          m
          (formatterFacesAt m facelist)
          (replicate m 0)
          (Some 0))
```

Appendix B

Induction Principles for Trees

We show that both definitions *tree* and *Tree*(see Section 3.2) are equivalent:

Lemma *tree-eq*: $((g,\, g') \in \mathit{tree}\ \mathit{succs}) = (g' \in \mathit{Tree}\ \mathit{succs}\ g)$

To this end, we first prove an induction principle and the reversed induction step rule for *tree*.

Lemma *tree-induct*:
$(h,\, h') \in \mathit{tree}\ \mathit{succs} \Longrightarrow$
$P\ h \Longrightarrow$
$(\bigwedge g\ g'.\ g' \in \mathit{set}\ (\mathit{succs}\ g) \Longrightarrow P\ g \Longrightarrow P\ g') \Longrightarrow$
$P\ h'$
proof −
assume *s*: $\bigwedge g\ g'.\ g' \in \mathit{set}\ (\mathit{succs}\ g) \Longrightarrow P\ g \Longrightarrow P\ g'$
assume $(h,\, h') \in \mathit{tree}\ \mathit{succs}\ P\ h$
then show $P\ h'$
proof (*induct rule*: *tree.induct*)
fix g **assume** $P\ g$ **then show** $P\ g$ **.**
next
fix $g\ g'\ g''$ **assume** $P\ g\ g' \in \mathit{set}\ (\mathit{succs}\ g)\ (g',\, g'') \in \mathit{tree}\ \mathit{succs}$
and *IH*: $P\ g' \Longrightarrow P\ g''$
have $P\ g'$ **by** (*rule s*)
then show $P\ g''$ **by** (*rule IH*)
qed
qed

Lemma *tree-rev-succs*:
$(g, g') \in \mathit{tree}\ \mathit{succs} \Longrightarrow g'' \in \mathit{set}\ (\mathit{succs}\ g') \Longrightarrow (g, g'') \in \mathit{tree}\ \mathit{succs}$
proof (*induct rule*: *tree.induct*)
fix g **assume** $g'' \in \mathit{set}\ (\mathit{succs}\ g)$
moreover have $(g'', g'') \in \mathit{tree}\ \mathit{succs}$ **by** (*rule tree.root*)
ultimately show $(g, g'') \in \mathit{tree}\ \mathit{succs}$ **by** (*rule tree.succs*)
next
fix $g\ h'\ h''$ **assume** $h' \in \mathit{set}\ (\mathit{succs}\ g)$
moreover assume $(h', h'') \in \mathit{tree}\ \mathit{succs}$ **and** $g'' \in \mathit{set}\ (\mathit{succs}\ h'')$
and *IH*: $g'' \in \mathit{set}\ (\mathit{succs}\ h'') \Longrightarrow (h', g'') \in \mathit{tree}\ \mathit{succs}$
have $(h', g'') \in \mathit{tree}\ \mathit{succs}$ **by** (*rule IH*)
ultimately show $(g, g'') \in \mathit{tree}\ \mathit{succs}$ **by** (*rule tree.succs*)
qed

Using these lemmas we can derive equivalence.

Lemma *tree-eq*: $((g, g') \in \mathit{tree}\ \mathit{succs}) = (g' \in \mathit{Tree}\ \mathit{succs}\ g)$
proof
assume $(g, g') \in \mathit{tree}\ \mathit{succs}$ **then show** $g \Rightarrow_{\mathit{succs}} g'$
proof (*induct rule*: *tree-induct*)
show $g \Rightarrow_{\mathit{succs}} g$ **by** (*rule Tree.root*)
next
fix $g'\ g''$ **assume** $g \Rightarrow_{\mathit{succs}} g'$ **and** $g' \rightarrow_{\mathit{succs}} g''$
show $g \Rightarrow_{\mathit{succs}} g''$ **by** (*rule Tree.succs*)
qed
next
assume $g \Rightarrow_{\mathit{succs}} g'$ **then show** $(g, g') \in \mathit{tree}\ \mathit{succs}$
proof (*induct rule*: *Tree.induct*)
show $(g, g) \in \mathit{tree}\ \mathit{succs}$ **by** (*rule tree.root*)
next
fix $g'\ g''$
assume $g \Rightarrow_{\mathit{succs}} g'$ **and** $(g, g') \in \mathit{tree}\ \mathit{succs}$ **and** $g'' \in \mathit{set}\ (\mathit{succs}\ g')$
show $(g, g'') \in \mathit{tree}\ \mathit{succs}$ **by** (*rule tree-rev-succs*)
qed
qed

Since we have proved that both definitions are equivalent, we can provide a second reversed induction principle for *Tree* and a reversed induction step rule.

Lemma *Tree-rev-induct*: $a \Rightarrow_{\mathit{succs}} b \Longrightarrow (\bigwedge g.\ P\ g\ g) \Longrightarrow$
$(\bigwedge g\ g'\ g''.\ g \rightarrow_{\mathit{succs}} g' \Longrightarrow g' \Rightarrow_{\mathit{succs}} g'' \Longrightarrow P\ g'\ g'' \Longrightarrow P\ g\ g'') \Longrightarrow$
$P\ a\ b$
by (*erule tree.induct* [*simplified tree-eq*]) *simp-all*

Lemma *Tree-rev-succs*:
$g' \Rightarrow_{succs} g'' \Longrightarrow (\bigwedge g.\ g \rightarrow_{succs} g' \Longrightarrow g \Rightarrow_{succs} g'')$
proof (*induct rule*: *Tree.induct*)
fix g **have** $g \Rightarrow_{succs} g$ **by** (*rule Tree.root*)
moreover assume $g \rightarrow_{succs} g'$
ultimately show $g \Rightarrow_{succs} g'$ **by** (*rule Tree.succs*)
next
fix $g\ g''\ g'''$
assume $g \rightarrow_{succs} g'$ **and** *IH*: $\bigwedge g.\ g \rightarrow_{succs} g' \Longrightarrow g \Rightarrow_{succs} g''$
have $g \Rightarrow_{succs} g''$ **by** (*rule IH*)
moreover assume $g'' \rightarrow_{succs} g'''$
ultimately show $g \Rightarrow_{succs} g'''$ **by** (*rule Tree.succs*)
qed

The next two lemmas are useful proof principles for the composition and subset of trees.

Lemma *Tree-compose*: $s \Rightarrow_{succs} g \Longrightarrow g \Rightarrow_{succs} g' \Longrightarrow s \Rightarrow_{succs} g'$
proof (*induct rule*: *Tree.induct*)
assume $s \Rightarrow_{succs} g'$ **then show** $s \Rightarrow_{succs} g'$ **.**
next
fix $h\ h'$ **assume** $h \rightarrow_{succs} h'$ **and** $h' \Rightarrow_{succs} g'$
and *IH*: $h \Rightarrow_{succs} g' \Longrightarrow s \Rightarrow_{succs} g'$
have $h \Rightarrow_{succs} g'$ **by** (*rule Tree-rev-succs*)
then show $s \Rightarrow_{succs} g'$ **by** (*rule IH*)
qed

Lemma *Tree-succs-subset*: $s \Rightarrow_{succs1} g \Longrightarrow$
$(\bigwedge g.\ s \Rightarrow_{succs2} g \Longrightarrow set\ (succs1\ g) \subseteq set\ (succs2\ g)) \Longrightarrow$
$s \Rightarrow_{succs2} g$
proof −
assume *succ*: $\bigwedge g.\ s \Rightarrow_{succs2} g \Longrightarrow set\ (succs1\ g) \subseteq set\ (succs2\ g)$
fix g **assume** $s \Rightarrow_{succs1} g$
then show $s \Rightarrow_{succs2} g$
proof (*induct rule*: *Tree.induct*)
case *root* **then show** *?case* **by** (*rule Tree.root*)
next
case (*succs* $g'\ g''$)
with *succs* **have** $g' \rightarrow_{succs2} g''$ **by** (*auto dest*: *succ*)
moreover from *succs* **have** $s \Rightarrow_{succs2} g'$ **by** *simp*
ultimately show $s \Rightarrow_{succs2} g''$ **by** (*rule-tac Tree.succs*)
qed
qed

Finally, we derive the following induction theorem for the terminal elements of a tree by induction on the construction.

Lemma $h \in terminalsTree\ seed\ succs \Longrightarrow$
$(\bigwedge param.\ P\ (seed\ param)) \Longrightarrow$
$(\bigwedge g\ g'\ param.\ g' \in set\ (succs\ param\ g) \Longrightarrow P\ g \Longrightarrow P\ g') \Longrightarrow P\ h$
proof −
assume r: $\bigwedge param.\ P\ (seed\ param)$
assume s: $\bigwedge g\ g'\ param.\ g' \in set\ (succs\ param\ g) \Longrightarrow P\ g \Longrightarrow P\ g'$
assume $h \in terminalsTree\ seed\ succs$
then obtain *param* **where**
$param$: $h \in terminalsTreeParam\ param\ seed\ succs$
by (*induct rule*: *terminalsTree.induct*) *auto*
then obtain *terminal*: $(seed\ param,\ h) \in tree\ (succs\ param)$
and *final* h **by** (*induct rule*: *terminalsTreeParam.induct*) *auto*
show $P\ h$
proof (*rule tree-induct*)
show $P(seed\ param)$ **by** (*rule* r)
fix $g\ g'$ **assume** $g' \in set\ (succs\ param\ g)\ P\ g$
then show $P\ g'$ **by** (*rule-tac* s)
qed
qed

Appendix C

Proof Texts

C.1 Lower Bound of Total Weight for Final Graphs

We present the proof sketch of the correctness proof of the lower bound for the total weight of an admissible weight function, as derived in Section 6.7. The main steps, indicated by (E_1), (E_2), (E_3), (A_1), (A_2), (A_3) correspond to the proof in Section 6.7.

Theorem *total-weight-lowerbound*:
$g \in PlaneGraphs_{8param} \Longrightarrow$
$admissible\ w\ g \Longrightarrow \sum_{f \in faces\ g} w\ f < squanderTarget \Longrightarrow$
$squanderLowerBound_{param}\ g \leq \sum_{f \in faces\ g} w\ f$
proof −
assume g: $g \in PlaneGraphs_{8param}$
then obtain p: $g \in PlaneGraphs$
then obtain *final*: $final\ g$
assume *admissible*: $admissible\ w\ g$
assume w: $\sum_{f \in faces\ g} w\ f < squanderTarget$

have $squanderLowerBound_{param}\ g$
$= ExcessNotAt_{param}\ g\ None + faceSquanderLowerBound_{param}\ g$

We expand the definition of *faceSquanderLowerBound*.

also have $faceSquanderLowerBound_{param}\ g$
$= \sum_{f \in faces\ g} \mathbf{d}\ |vertices\ f|$

We expand the definition of *ExcessNotAt*.

also obtain V **where** *eq*: $ExcessNotAt_{param}\ g\ None$
$= \sum_{v\ \in\ V} ExcessAt_{param}\ g\ v$
and *pS*: $preSeparated\ g\ (set\ V)$
and *V-subset*: $set\ V \subseteq set(vertices\ g)$
and *V-distinct*: $distinct\ V$

We partition V in two disjoint subsets $V1, V2$, where $V2$ contains all exceptional vertices, $V1$ all not exceptional vertices.

also def $V1 \equiv [v \in V.\ except\ g\ v = 0]$
def $V2 \equiv [v \in V.\ except\ g\ v \neq 0]$

have $(\sum_{v\ \in\ V} ExcessAt_{param}\ g\ v)$
$= (\sum_{v\ \in\ V1} ExcessAt_{param}\ g\ v) + (\sum_{v\ \in\ V2} ExcessAt_{param}\ g\ v)$

We partition $V2$ in two disjoint subsets, $V4$ contains all exceptional vertices of degree $\neq 5$ $V3$ contains all exceptional vertices of degree 5.

also def $V4 \equiv [v \in V2.\ degree\ g\ v \neq 5]$
def $V3 \equiv [v \in V2.\ degree\ g\ v = 5]$

have $(\sum_{v\ \in\ V2} ExcessAt_{param}\ g\ v)$
$= (\sum_{v\ \in\ V3} ExcessAt_{param}\ g\ v) + \sum_{v\ \in\ V4} ExcessAt_{param}\ g\ v$

We partition *faces g* in two disjoint subsets: $F1$ contains all faces that contain a vertex of $V1$, $F2$ the remaining faces.

also def $F1 \equiv [f \in faces\ g\ .\ \exists\ v \in set\ V1.\ f \in set\ (facesAt\ g\ v)]$
def $F2 \equiv [f \in faces\ g\ .\ \neg(\exists\ v \in set\ V1.\ f \in set\ (facesAt\ g\ v))]$

have $\sum_{f\ \in\ faces\ g}$ **d** $|vertices\ f|$
$= (\sum_{f\ \in\ F1}$ **d** $|vertices\ f|\ ) + \sum_{\ f\ \in\ F2}$ **d** $|vertices\ f|$

We split up $F2$ in two disjoint subsets:

also def $F3 \equiv [f \in F2.\ \exists\, v \in set\ V3.\ f \in set\ (facesAt\ g\ v)]$
def $F4 \equiv [f \in F2.\ \neg\ (\exists\, v \in set\ V3.\ f \in set\ (facesAt\ g\ v))]$

have *F3*: $F3 = [f \in faces\ g\ .\ \exists\, v \in set\ V3.\ f \in set\ (facesAt\ g\ v)]$
have $(\sum_{f \in F2}$ **d** $|vertices\ f|\ )$
$= (\sum_{f \in F3}$ **d** $|vertices\ f|\ ) + \sum_{f \in F4}$ **d** $|vertices\ f|$

(E_1) From the definition of *ExcessAt* we have

also have $(\sum_{v \in V1} ExcessAt_{param}\ g\ v) + \sum_{f \in F1} \mathbf{d}\ |vertices\ f| = \sum_{v \in V1} \mathbf{b'}_{param}\ (tri\ g\ v)\ (quad\ g\ v)$
proof −
have $\sum_{f \in F1} \mathbf{d}\ |vertices\ f| = \sum_{v \in V1} (tri\ g\ v * \mathbf{d}\ 3 + quad\ g\ v * \mathbf{d}\ 4)$
also have $(\sum_{v \in V1} ExcessAt_{param}\ g\ v) + \sum_{v \in V1} (tri\ g\ v * \mathbf{d}\ 3 + quad\ g\ v * \mathbf{d}\ 4) = \sum_{v \in V1} (ExcessAt_{param}\ g\ v + tri\ g\ v * \mathbf{d}\ 3 + quad\ g\ v * \mathbf{d}\ 4)$
also have $\ldots = \sum_{v \in V1} \mathbf{b'}_{param}\ (tri\ g\ v)\ (quad\ g\ v)$
finally show *?thesis* **.**
qed

(E_2) For all exceptional vertices of degree 5 *excess* returns *a* (*tri g v*).

also from *p final V-subset* **have** $(\sum_{v \in V3} ExcessAt_{param}\ g\ v) = \sum_{v \in V3} \mathbf{a}\ (tri\ g\ v)$

(E_3) For all exceptional vertices of degree $\neq 5$ *ExcessAt* returns 0.

also from *p final* **have** $(\sum_{v \in V4} ExcessAt_{param}\ g\ v) = \sum_{v \in V4} 0$

also have $\ldots = 0$

(A_1) We use property $admissible_2$.

also have
$\sum_{v \in V1} \mathbf{b'}_{param}\ (tri\ g\ v)\ (quad\ g\ v) \leq (\sum_{v \in V1} \sum_{f \in facesAt\ g\ v} w\ f)$

proof (*rule-tac ListSum-le*)
fix *v* **assume** *v* ∈ *set V1*
with *V1-def V-subset* **have** *v* ∈ *set* (*vertices g*)
with *admissible* **show** $\mathbf{b'}_{param}\ (tri\ g\ v)\ (quad\ g\ v) \leq \sum_{f \in facesAt\ g\ v} w\ f$
qed

also from *pSV1 V1-distinct* **have** $\ldots = \sum_{f \in F1} w\ f$

(A_2) We use property $admissible_4$.

also from *admissible V3 V3-subset* **have**
$(\sum_{v \in V3} \mathbf{a}\ (tri\ g\ v)) + (\sum_{f \in F3} \mathbf{d}\ |vertices\ f|\ ) \leq \sum_{f \in F3} w\ f$

(A_3) We use property *admissible*$_1$.

also have $\sum_{f \in F4}$ **d** $|vertices\ f| \leq \sum_{f \in F4}\ w\ f$
proof (*rule ListSum-le*)
fix f **assume** $f \in set\ F4$
then have f: $f \in set\ (faces\ g)$
with *admissible*$_1$ f **show d** $|vertices\ f| \leq w\ f$
qed

We reunite $F3$ and $F4$.

also have $(\sum_{f \in F3}\ w\ f) + (\sum_{f \in F4}\ w\ f) = (\sum_{f \in F2}\ w\ f)$

We reunite $F1$ and $F2$.

also have $(\sum_{f \in F1}\ w\ f) + (\sum_{f \in F2}\ w\ f) = \sum_{f \in faces\ g}\ w\ f$

finally show $squanderLowerBound_{param}\ g \leq \sum_{f \in faces\ g}\ w\ f$.
qed

C.2 Enumeration

The following property is used in the completeness proof of the refinement step of neglecting final graphs: *Enumeration* contains exactly the graphs that are not neglectable final graphs.

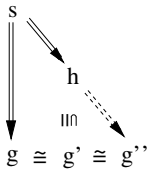
Lemma *PlaneGraphs*$_8$*Param-eq*: *EnumerationParam param*
$= \{g.\ g \in PlaneGraphs_{8param} \wedge \neg\ neglectableFinal_{param}\ g\ \}$ (**is** *?L* = *?R*)
proof(*intro equalityI subsetI conjI CollectI*)
fix g **assume** *L*: $g \in$ *?L*
then have f: *final g*
from *L* **have** g: $g \in EnumerationTree\ param$

then have $g \in PlaneGraphs_8 Tree_{param}$
proof
fix g
assume g: $g \in Tree\ (succsPlane_{8}\ param)\ Seed_{param}$
then have g: $g \in Tree\ succsPlane_5\ param\ Seed_{param}$
show $set\ (succsEnumeration_{param}\ g) \subseteq set\ (succsPlane_{8}\ param\ g)$
proof
fix g' **assume** g': $g' \in set\ (succsEnumeration_{param}\ g)$
then have $g' \in set\ (succsPlane_{5}\ param\ g)$

```
      with g have g'5: g' ∈ Tree succsPlane_5 param Seed_param
        by (rule-tac Tree.succs)
      from g' show g' ∈ set (succsPlane_8 param g)
      proof cases
        assume final g'
        with g'5 have g' ∈ PlaneGraphs_5param
        with g' show ?thesis
      next
        assume ¬ final g' with g' show ?thesis
      qed
    qed
  qed
  then show g ∈ PlaneGraphs_8param
  from g show  ¬ neglectableFinal_param g
next
  fix g assume R: g ∈ ?R
  then have f: final g
  from R show g ∈ ?L
  proof     assume g: g ∈ PlaneGraphs_8param ¬ neglectableFinal_param g
    then have g ∈ PlaneGraphs_8 Tree_param
    then have g ∈ Tree succsPlane_8 param Seed_param
    moreover assume n: ¬ neglectableFinal_param g
    ultimately have  g ∈ Tree succsEnumeration_param Seed_param
    proof (induct rule: Tree.induct)
      case root then show ?case by (rule-tac Tree.root)
    next
      case (succs g' g'')
      then have s: ¬ neglectableFinal_param g''
        g'' ∈ set (succsPlane_8 param g')
      then have ¬ neglectableFinal_param g'
      then have g' ∈ Tree succsEnumeration_param Seed_param
        by (rule-tac succs)
           moreover
      from s have g'' ∈ set (succsEnumeration_param g')

      ultimately  show ?case by (rule-tac Tree.succs)
    qed
    then have g ∈ EnumerationTree param
    with f show ?thesis
  qed
qed
```

C.3 Reordering of Faces

We sketch a proof of a property about reorderings of faces for the case of a common initial graph.

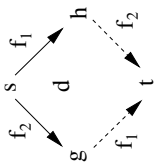


Theorem *subset-completion*: $h \in$ *Tree* (*successorsList*$_{param}$) $s \Longrightarrow$
$g \in$ *Tree* (*successorsList*$_{param}$) $s \Longrightarrow$
set (*finals* h) $\subseteq_{\cong}$ *set* (*finals* g) $\Longrightarrow$
$\exists\, g'.\ g' \cong g \wedge g' \in$ *Tree* (*successorsList*$_{param}$) h

We need the following basic property of the successor function, to prove the theorem *subset-completion* in two different successor graphs a different new face is added (up to isomorphism).

Lemma e: $g \in$ *set* (*successorsList*$_{param}$ s) $\Longrightarrow$
$h \in$ *set* (*successorsList*$_{param}$ s) $\Longrightarrow$
set (*finals* h) $= \{f_1\} \cup$ *set* (*finals* s) $\Longrightarrow f_1 \notin_{\cong}$ *set* (*finals* s) $\Longrightarrow$
set (*finals* g) $= \{f_2\} \cup$ *set* (*finals* s) $\Longrightarrow f_2 \notin_{\cong}$ *set* (*finals* s) $\Longrightarrow$
$f_1 \cong f_2 \Longrightarrow g = h$

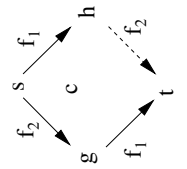
The second lemma states that we can transpose the addition of two faces under certain conditions: if we can add a final face f_1 and a final face f_2 in a graph s then we can do this in both orders, provided that these faces do not interfere. For example, their edges are not allowed to intersect. This is ruled out by the precondition that both faces are faces of a certain partial plane graph p.



Lemma d: $\neg\, f_1 \cong f_2 \Longrightarrow \{f_1, f_2\} \cup$ *set* (*finals* s) $\subseteq_{\cong}$ *set* (*finals* p) $\Longrightarrow$
$s \rightarrow_{successorsList\ param} h \Longrightarrow$
set (*finals* h) $= \{f_1\} \cup$ *set* (*finals* s) $\Longrightarrow f_1 \notin_{\cong}$ *set* (*finals* s) $\Longrightarrow$

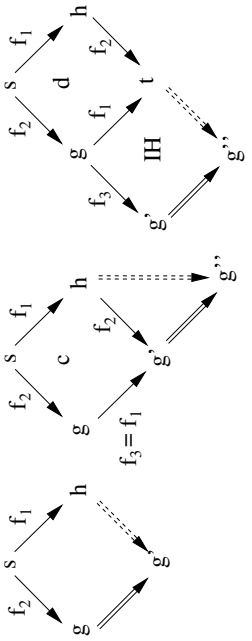
$s \rightarrow_{successorsList\ param} g \Longrightarrow$
$set\ (finals\ g) = \{f_2\} \cup set\ (finals\ s) \Longrightarrow f_2 \notin_{\cong} set\ (finals\ s) \Longrightarrow$
$\exists\, t.\ g \rightarrow_{successorsList\ param} t \wedge h \rightarrow_{successorsList\ param} t$
$\wedge\ set\ (finals\ t) =_{\cong} \{f_1\} \cup set\ (finals\ g)$

The next lemma expresses a similar property: if we can add a face f_2 followed by a face f_1 and could also first add f_1, then we can add f_2 after f_1.



Lemma c: $h \in set\ (successorsList_{param}\ s) \Longrightarrow$
$set\ (finals\ h) = \{f_1\} \cup set\ (finals\ s) \Longrightarrow f_1 \notin_{\cong} set\ (finals\ s) \Longrightarrow$
$g \in set\ (successorsList_{param}\ s) \Longrightarrow$
$set\ (finals\ g) = \{f_2\} \cup set\ (finals\ s) \Longrightarrow f_2 \notin_{\cong} set\ (finals\ s) \Longrightarrow$
$t \in set\ (successorsList_{param}\ g) \Longrightarrow$
$set\ (finals\ t) = \{f_1\} \cup set\ (finals\ g) \Longrightarrow f_1 \notin_{\cong} set\ (finals\ g) \Longrightarrow$
$t \in set\ (successorsList_{param}\ h)$

Now we combine the properties c, d and e and prove the following lemma by induction on the construction of a tree.



Lemma *b*: $g' \in Tree\ (successorsList_{param})\ g \Longrightarrow$
$(\bigwedge h\ s.\ g \in set\ (successorsList_{param}\ s) \Longrightarrow$
$h \in set\ (successorsList_{param}\ s) \Longrightarrow$
$set\ (finals\ h) \subseteq\cong set\ (finals\ g') \Longrightarrow$
$g' \in Tree\ (successorsList_{param})\ h)$
proof (*induct rule*: *Tree-rev-induct*)
case (*root g*) **then show** *?case*
next
case (*succs g g' g''*)
have *h*: $h \in set\ (successorsList_{param}\ s)$ **.**
then obtain f_1 **where** *fh*: $set\ (finals\ h) = \{f_1\} \cup set\ (finals\ s)$
$f_1 \notin\cong set\ (finals\ s)$
by (*auto dest*: *successorsList-newfinal*)
have *g*: $g \in set\ (successorsList_{param}\ s)$ **.**
then obtain f_2 **where** *fg*: $set\ (finals\ g) = \{f_2\} \cup set\ (finals\ s)$
$f_2 \notin\cong set\ (finals\ s)$
by (*auto dest*: *successorsList-newfinal*)
have *g'*: $g' \in set\ (successorsList_{param}\ g)$ **.**
then obtain f_3 **where** *fg'*: $set\ (finals\ g') = \{f_3\} \cup set\ (finals\ g)$
$f_3 \notin\cong set\ (finals\ g)$
by (*auto dest*: *successorsList-newfinal*)

show $g'' \in Tree\ (successorsList_{param})\ h$
proof (*cases* $f_1 \cong f_2$)
case *True*
then have *eq*: $g = h$ **by** (*rule-tac e*)
have $g'' \in Tree\ (successorsList_{param})\ g$ **by** (*rule Tree-rev-succs*)
with *eq* **show** *?thesis* **by** *auto*
next
case *False*
note $n = this$
then show *?thesis*
proof (*cases* $f_3 = f_1$)
case *True*
from *h fh g fg g' fg'*[*simplified True*]
have $g' \in set\ (successorsList_{param}\ h)$
by (*rule c*)

then show $g'' \in Tree\ (successorsList_{param})\ h$ **by** (*rule-tac Tree-rev-succs*)
next
case *False*
from *succs* **have** *hyp*: $set\ (finals\ h) \subseteq\cong set\ (finals\ g'')$ **by** *simp*
from *succs* **have** $set\ (finals\ g) \subseteq set\ (finals\ g')$

by (*simp add*: *successorsList-finals*)
also from *succs* **have** *set* (*finals* g') $\subseteq$ *set* (*finals* g'')
by (*simp add*: *Tree-finals*)
finally have *set* (*finals* g) $\subseteq$ *set* (*finals* g'') **.**

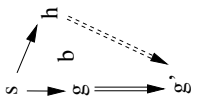
moreover then have f_2: $f_2 \in$ *set* (*finals* g'')
by (*simp add*: *fg successorsList-finals*)

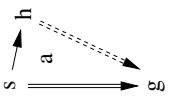
from *succs fh* **have** f_1: $f_1 \in\cong$ *set* (*finals* g'')
by (*auto simp add*: *iso-eqs*)

moreover from f_1 f_2 *fh hyp* **have**
$\{f_1, f_2\} \cup$ *set* (*finals* s) $\subseteq\cong$ *set* (*finals* g'')
by (*auto simp add*:*iso-eqs*)
with n h g *fh fg*
obtain t **where** $g \rightarrow_{successorsList\ param} t$
$h \rightarrow_{successorsList\ param} t$
set (*finals* t) $=\cong \{f_1\} \cup$ *set* (*finals* g)
by (*auto dest*: d)

ultimately have *set* (*finals* t) $\subseteq\cong$ *set* (*finals* g'')
by (*erule-tac* r) *auto*

then have $g'' \in$ *Tree* ($successorsList_{param}$) t
by (*rule-tac succs*)
then show $g'' \in$ *Tree* ($successorsList_{param}$) h
by (*rule-tac Tree-rev-succs*)
qed
qed
qed





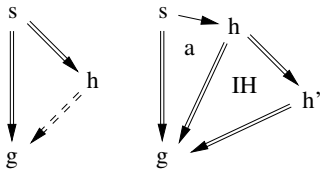
Lemma a: $g \in$ *Tree* ($successorsList_{param}$) $s \Longrightarrow$
$h \in$ *set* ($successorsList_{param}$ s) $\Longrightarrow$

set (*finals h*) $\subseteq_{\cong}$ *set* (*finals g*) $\Longrightarrow$
g $\in$ *Tree* (*successorsList*$_{param}$) *h*
proof (*induct rule*: *Tree-rev-induct*)
case (*root s*)
have *h* $\in$ *set* (*successorsList*$_{param}$ *s*) **.**
then have *set* (*finals s*) $\subset_{\cong}$ *set* (*finals h*)
by (*simp add*: *successorsList-subset*)
with *root* **show** *?case* **by** (*auto simp add*:*iso-eqs*)
next
let *?succs* = *successorsList*$_{param}$
case (*succs s g g'*)

have *g* $\in$ *set* (*successorsList*$_{param}$ *s*)
g' $\in$ *Tree* (*successorsList*$_{param}$) *g*
h $\in$ *set* (*successorsList*$_{param}$ *s*)
set (*finals h*) $\subseteq_{\cong}$ *set* (*finals g'*) **.**

then show *g'* $\in$ *Tree* (*successorsList*$_{param}$) *h*
by (*rule-tac b*)
qed

We finally prove the theorem *subset-completion*, which shows that the order, in which the faces of a graph are constructed, can be changed, as long as the graph stays connected during the construction.



Lemma *subset-completion*: *h* $\in$ *Tree* (*successorsList*$_{param}$) *s* $\Longrightarrow$
g $\in$ *Tree* (*successorsList*$_{param}$) *s* $\Longrightarrow$
set (*finals h*) $\subseteq_{\cong}$ *set* (*finals g*) $\Longrightarrow$
$\exists\, g'.\ g' \cong g \wedge g' \in$ *Tree* (*successorsList*$_{param}$) *h*
proof (*induct rule*: *Tree-rev-induct*)
case (*root s*) **then show** *?case* **by** *auto*
next
let *?succs* = *successorsList*$_{param}$
case (*succs s h h'*)
then have *set* (*finals h*) $\subseteq$ *set* (*finals h'*) **by** (*simp add*: *Tree-finals*)

also have *set* (*finals* h') $\subseteq_{\cong}$ *set* (*finals* g) **.**
finally have *set* (*finals* h) $\subseteq_{\cong}$ *set* (*finals* g) **.**

moreover have $h \in$ *set* ($successorsList_{param}$ s)
$g \in$ *Tree* ($successorsList_{param}$) s **.**
ultimately have $g \in$ *Tree* ($successorsList_{param}$) h **by** (*rule-tac* a)
then show $\exists\, g'.\ g' \cong g \wedge g' \in$ *Tree* ($successorsList_{param}$) h'
by (*rule succs*)
qed

Appendix D

Statistics

We summarize the numbers of graphs generated for each parameter and the execution times on a 3 GHz Xeon processor.

QuadParameter				
parameter	partial Graphs	final Graphs	isomorphic	time
0	1078	1		3.5 secs
1	1497	1		8.0 secs
2	2766	15		61.7 secs
3	1193	12		5.0 secs
4	3253	50		20.3 secs
5	23374	314		240.2 secs
6	24395	317		281.0 secs
7	1628	248		38.8 secs
8	3865	97		37.6 secs
9	263	35		8.9 secs
10	111483	3317		4198.7 secs
11	2505	312		75.4 secs
12	4354	568		116.4 secs
13	1141	188		64.6 secs
14	618	5		6.2 secs
15	554	5		23.3 secs
16	30	1		1.3 secs
quad	183997	5486	981	5191.7 secs

ExceptionalParameter				
parameter	partial Graphs	final Graphs	isomorphic	time
5	739599	4729	1779	2713.4 secs
6	279497	741	245	375.8 secs
7	46151	67	23	70.8 secs
8	50232	74	22	93.2 secs
total	1299473	11097	3050	8444.9 secs

Bibliography

[1] K. Appel and W. Haken. Solution of the four color map problem. *Scientific American*, 237(4):108–121, October 1977.

[2] K. Appel and W. Haken. Every planar map is four colorable. *Contemporary Mathematics*, 98, 1998.

[3] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. *Illinois: Journal of Mathematics*, 21:439–567, December 1977.

[4] G. Bauer and T. Nipkow. The 5 Colour Theorem in Isabelle/Isar. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs'2002, Hampton, VA, USA*, volume 2410 of *LNCS*. Springer, 2002.

[5] G. Bauer and T. Nipkow. Towards a Verified Enumeration of all Tame Plane Graphs. In T. Coquand, H. Lombardi, and M.-F. Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. http://www.dagstuhl.de/files/Proceedings/05/05021/05021.NipkowTobias.Paper.pdf.

[6] S. Berghofer and T. Nipkow. Executing Higher Order Logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lect. Notes in Comp. Sci.*, pages 24–40. Springer-Verlag, 2002.

[7] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In J. R. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.

[8] B. Bollobás. *Modern Graph Theory*. Graduate Texts in Mathematics. Springer, New York, 1998.

[9] The coq proof assistant. http://coq.inria.fr/.

[10] R. Cori and A. Machi. Maps, hypermaps and their automorphisms: a survey, I,II,III. *Expositiones Mathematicae*, 10:403–467, 1992.

[11] S. P. Ferguson. Sphere packings. v. http://arxiv.org/abs/math.MG/9811077/, 1998.

[12] S. P. Ferguson and T. C. Hales. A formulation of the kepler conjecture. http://www.math.pitt.edu/~thales/kepler98/form.ps, 1998.

[13] G. Gonthier. A computer-checked proof of the four colour theorem. http://research.microsoft.com/~gonthier/4colproof.pdf, 2005.

[14] T. C. Hales. The flyspeck project fact sheet. http://www.math.pitt.edu/~thales/flyspeck/.

[15] T. C. Hales. Sphere packings. i. *Disc. Comput. Geom.*, 17:1–51, 1997. http://www.math.pitt.edu/~thales/kepler98/sphere1.ps.

[16] T. C. Hales. Sphere packings. ii. *Disc. Comput. Geom.*, 18:135–149, 1997. http://www.math.pitt.edu/~thales/kepler98/sphere2.ps.

[17] T. C. Hales. The kepler conjecture. http://www.math.pitt.edu/~thales/kepler98/, 1998.

[18] T. C. Hales. An overview of the kepler conjecture. http://www.math.pitt.edu/~thales/kepler98/sphere0.ps, 1998.

[19] T. C. Hales. Sphere packings. iii. http://www.math.pitt.edu/~thales/kepler98/sphere3.ps, 1998.

[20] T. C. Hales. Sphere packings. iv. http://www.math.pitt.edu/~thales/kepler98/sphere4.ps, 1998.

[21] T. C. Hales. Sphere packings. vi. http://www.math.pitt.edu/~thales/kepler98/sphere6.ps, 1998.

[22] T. C. Hales. Cannonballs and honeycombs. *Notices of the American Maths Soc*, 47(4):440–449, 2000.

[23] T. C. Hales. The Kepler Conjecture, 2002. Complete sources of the Java program used in the proof of the Kepler conjecture.

[24] T. C. Hales. A Proof of the Kepler Conjecture. http://www.math.pitt.edu/~thales/kepler04/fullkepler.pdf, 2004.

[25] T. C. Hales and S. P. Ferguson. A Proof of the Kepler Conjecture. *Ann. Math.*, 2005. to appear.

[26] C. Holden. Stacking up the evidence. *Science*, 299:1513, 2003.

[27] D. Mackenzie. What in the Name of Euclid Is Going On Here? *Science*, 307:1402–1403, 2005.

[28] T. Nipkow. Reduced Archive of Tame Plane Graphs, 2005. http://www.in.tum.de/~nipkow/Flyspeck/.

[29] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.

[30] N. Robertson, D. Sanders, P. Seymour, and R. Thomas. The four color theorem. http://www.math.gatech.edu/~thomas/FC/fourcolor.html.

[31] G. Szpiro. Does the proof stack up? *Nature*, 424:12–13, 2003.

[32] R. Thomas. An update on the four-color theorem. http://www.ams.org/notices/199807/thomas.pdf, August 1998.

[33] W. Tutte. *Graph Theory*. Cambridge University Press, 2001.

[34] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.

Index