Towards a Verified Enumeration of All Tame Plane Graphs

Gertrud Bauer, Tobias Nipkow

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
www.in.tum.de/~bauerg, www.in.tum.de/~nipkow

Abstract. We contribute to the fully formal verification of Hales' proof of the Kepler Conjecure by analyzing the enumeration of all *tame* plane graphs. We sketch a formalization of plane graphs, tameness and Hales' enumeration procedure in Higher Order Logic. The correctness of the enumeration is partially verified (which uncovered a small mismatch between Hales' definition of tameness and his enumeration procedure). By executing the enumeration in ML we confirm that a list of plane graphs provided by Hales (the *archive*) contains all tame plane graphs (although it also contains much redundancy).

1 Introduction

In 1611, Kepler proposed that the cubic close packing (see fig. 1) the and hexagonal close packing (both of which have maximum densities of $\frac{\pi}{3\sqrt{2}} \approx 74.048\%$) are the densest possible sphere packings.

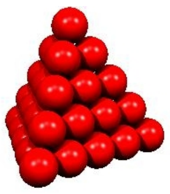


Fig. 1. Face centered cubic packing

Theorem (Kepler Conjecture).
The face-centered cubic packing of unit spheres has optimal density.

In 1998, Thomas C. Hales announced to have found a proof of the Kepler conjecture using a computer program [2]. After 5 years the referees stated that they were 99% certain of the correctness of the proof. Hence Hales initiated the flyspeck project with the aim to produce a formal proof of the Kepler Conjecture.

Hales' proof relies on a notion of *tame* plane graphs and a (finite) list of all tame plane graphs, the *archive*.

Proof Structure of Kepler Conjecture (by contradiction)

1. Assume there is a counterexample.
2. Associate a plane graph ('contravening graph') to every counterexample.
3. **Theorem.** Every contravening graph is tame.
4. **Theorem 1.** Every tame plane graph is isomorphic to a graph in the archive.
5. For every graph in the archive show that it is not contravening (using linear programs and more).

Qed.

The aim of our work is the verification of Theorem 1, that is a proof of the theorem

Theorem $g \in \mathit{Planes} \Longrightarrow \mathit{tame}\ g \Longrightarrow \exists\, h \in \mathit{archive}.\ g \cong h$

Hales created the archive from the output of a Java program he wrote for generating all tame plane graphs. Hence the crux of the proof of this theorem is the completeness of his Java program (modulo graph isomorphism).

Our contributions are

– A Higher Order Logic version (*enum*) of Hales' Java program, generating all tame plane graphs.
– A partial proof of completeness of *enum*.
– A confirmation of Theorem 1 by executing *enum* and checking that its output is contained in the archive.
– A reduced archive.

All formalizations and proofs are carried out in Higher Order Logic (HOL) with the theorem prover Isabelle [5].

As a 'side effect' of the verification it turned out that the definition of tame graphs in the proof of the Kepler conjecture needs to be changed a bit (see §4.4). However, the correctness of the Kepler conjecture is not affected by this change.

Notations We use the following Isabelle/HOL notations for Lists and Sets: Given a list xs, $\mathit{hd}\ xs$ is the first element in xs, $\mathit{last}\ xs$ is the last element xs, $\mathit{rev}\ xs$ is the reverted list, $x\#xs$ appends an element x in front of xs, $xs[\![i]\!]$ is element at position i in xs, $|xs|$ is the length of xs, $[x \in xs.\ P\ x]$ the list of all elements of xs that obey property P, $\mathit{replicate}\ n\ x$ is the list containing n times the element x, $\mathit{set}\ xs$ is the set of elements in xs, $[0\ ..<\ \mathit{int}\ n]$ is the list of integers from 0 to $n - 1$, $\{f\ x \mid x.\ x \in xs\}$ is the set of all elements $f\ x$ with $x \in xs$.

Structure of this paper The structure of this paper is as follows: In §2 we introduce the data structures and some functions for faces and graphs in Isabelle/HOL. In §3 we show the definition of plane graphs and their implementation in Isabelle/HOL. In §4 we recall the definition of tameness and show its translation to Isabelle/HOL. In §5 we show the proof structure of the completeness theorem induced by a set of refinement steps of the enumeration algorithm. In §6 we show the definition of graph isomorphism and an executable isomorphism test.

2 Graphs

In this section we define plane graphs in terms of sets of faces.

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

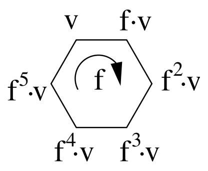


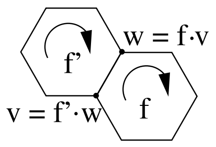
Fig. 2. Example: A Face of length 6

When we want to draw graphs we need a convention in which orientation faces are to be drawn. We draw faces in clockwise orientation (see fig. 2).

An *unoriented edge* is a two-element set $\{v, w\}$ of vertices such that $f{\cdot}v = w$ for some face f. The vertices v and w are then said to be adjacent. An *(oriented) edge* is a pair (v, w) such that $f{\cdot}v = w$ for some face f.

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

1. If $f{\cdot}v = w$ then there is a unique face f' in g, with $f'{\cdot}w = v$. Hence associated to each vertex v in a graph g there is an automorphism of the faces of g that contain v. We write $f \mapsto (g, v){\cdot}f = f'$ for this function. Hence $v = (g, v){\cdot}f{\cdot}w$. Moreover, each 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.

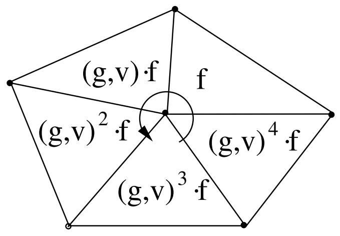


2. For each vertex, the function $f \mapsto (g, v){\cdot}f$ is a cyclic permutation of the set of faces containing v.

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

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

Note that properties 1 and 2 together with the convention of drawing the vertices in a face in clockwise orientation imply that $f \mapsto (g, v) \cdot f$ permutes the faces around a vertex v in counterclockwise orientation.



The *length* $|f|$ is the number of vertices in a face f. A face of length 3 is called *triangle*, a face of length 4 is called *quadrilateral*, faces of length at least 5 are called *exceptional*. The degree of a vertex is the number of faces containing the vertex. $\mathit{tri}(v)$ is the number of triangles containing a vertex v. $\mathit{quad}(v)$ is the number of quadrilaterals containing a vertex v. $\mathit{except}(v)$ 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 $\mathit{type}(v) = (p, q)$ for $\mathit{type}(v) = (p, q, 0)$.

Two graphs g_1 and g_2 are called *properly isomorphic*, 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 a graph g_2 ($g_1 \cong g_2$), if g_1 is properly 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, whereas in Hales' paper [3] faces are called complete/incomplete). In our figures we draw final faces *white* and nonfinal faces (except a nonfinal face at the outside of a graph) *grey*.

Final faces are those that occur in the terminal graph, nonfinal faces can be further refined by adding new faces.

2.1 Representation of Faces

We use integers as identifiers for vertices.

types *vertex* = *int*

We represent faces by lists of (distinct) vertices and a face type. The type of a face is either final or nonfinal.

datatype *facetype* = *Final* | *Nonfinal*

datatype *face* = *Face* (*vertex list*) *facetype*

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

The set of vertices V_f in a face f is denoted by *set* (*vertices f*) where

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

The function *nextVertex* (written as $f{\cdot}v$), is based on *findNext*, which returns the successor of an element in a list.

```
findNext :: 'a list ⇒ 'a ⇒ 'a ⇒ 'a
findNext [] b c = c
 findNext (a#as) b c =
   (case as of [] ⇒  c
          | (d#ds) ⇒ if a = b then d else findNext as b c)
```

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

Note that the function *nextVertex* will also return a value if v is not a vertex of *f*, because in Isabelle/HOL all functions have to be total. But during the generation the function is only called on vertices of *f*. For the verification of this function a precondition is introduced.

The *inverse* face f^{-1} of a face f is obtained by reversing the cyclic order in f. We define the permutation function $f^{-1} \cdot v$ of the inverse face f^{-1}.

```
prevVertex :: face ⇒ vertex ⇒ vertex
 f⁻¹ · v ≡ (let vs = vertices f in findNext (rev vs) v (last vs))
```

2.2 Representation of Graphs

A graph g is implemented as a datatype with the following components:

- *vertices g*, the list of vertices in g.
- *faces g*, the list of faces in g.
- *countVertices g*, the number of vertices in g.
- *faceListAt g*, an incidence list of face lists, assigning each vertex v a list of faces in g containing v.
- *heights g*, a list of integers, assigning each vertex an integer height.
- *baseVertex g*, one of the vertices.

Note that for efficiency reasons of the enumeration algorithm, the representation is highly redundant, for example *vertices g* could be calculated from *faces g*, *countVertices g* could be calculated from *vertices g*, and *faceListAt g* could be calculated from *faces g*. The last two components are only used for optimization of the generation process.

Moreover it allows graphs that are not well-formed (inconsistent). Well-formedness is guaranteed by the inductive definition of plane graphs.

types $faces = face\ list\ list$
types $heights = int\ list$

datatype $graph = Graph\ (vertex\ list)\ (face\ list)\ int\ faces\ heights\ (vertex\ option)$

$vertices\ (Graph\ vs\ fs\ n\ f\ h\ b) = vs$

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

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

$faceListAt :: graph \Rightarrow faces$
$faceListAt\ (Graph\ vs\ fs\ n\ f\ h\ b) = f$

$facesAt :: graph \Rightarrow vertex \Rightarrow face\ list$
$facesAt\ g\ v \equiv faceListAt\ g\ [\![v]\!]$

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

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

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

A graph is final if all faces are final.

$finals :: graph \Rightarrow face\ list$
$finals\ g \equiv [f \in faces\ g.\ final\ f]$

$nonFinals :: graph \Rightarrow face\ list$
$nonFinals\ g \equiv [f \in faces\ g.\ \neg\ final\ f]$

$final\ g \equiv (nonFinals\ g = [])$

A vertex is final if all incident faces are final.

$nonFinalsAt :: graph \Rightarrow vertex \Rightarrow face\ list$
$nonFinalsAt\ g\ v \equiv [f \in facesAt\ g\ v.\ \neg\ final\ f]$

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

2.3 Counting the Number of Faces at a Vertex

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

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

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

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

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

An edge (a, b) is contained in face f, if b is the successor of a in f.

$edges :: {}'a \Rightarrow (vertex \times vertex)\ set$

$edges\ (f{::}face) \equiv \{(a,\ f \cdot a)|a.\ a \in set\ (vertices\ f)\}$
$edges\ (g{::}graph) \equiv \bigcup f \in set\ (faces\ g).\ edges\ f$

The function *nextFace* (written as $f \mapsto (g, v){\cdot}f$) permutes the faces at a vertex *v*.

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

3 Plane Graphs

We use an inductive definition of (connected) plane graphs.

We start the construction with *initial* graphs (*seed graphs*): Note that the set of faces allways contains also the outer face. Hence initial graphs consist of two faces, one final inner and one nonfinal outer, where the final one is the inverse of the nonfinal one.

We modify a graph g by adding a new final face in a nonfinal face f of g. This is performed by applying a *patch* for f at an edge e of f. A patch p is a partial graph with at least 2 faces, one final face f_2 (the final face we want to add to the graph), one nonfinal face f_1 (the inverse face of f) and 0 or more nonfinal faces, uniquely determined by f and f_2, filling the 'gap' between the old nonfinal face f and the new final face f_2 (see example in fig. 3).

We apply a patch by replacing f in g by $p - \{f_1\}$. The new nonfinal faces complete the new graph such that again every edge is contained in exactly two faces (in opposite directions).

Ultimately, we define a partial plane graph inductively as follows:

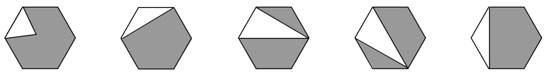


Fig. 3. All patches for a face of length 6 with final face of length 3

(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. Every final plane graph can be identified with a plane graph.

It is easy to show that every graph generated using this inductive definition is plane. This can be verified by induction over the generation, using Euler's formula. On the other hand it must be shown that we can reach every plane graph using this construction.

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 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 thorem 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 fig.4). 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.

3.1 Construction of Seed Graphs

An *initial 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 n*.

$graph :: nat \Rightarrow graph$
$graph\ n \equiv$

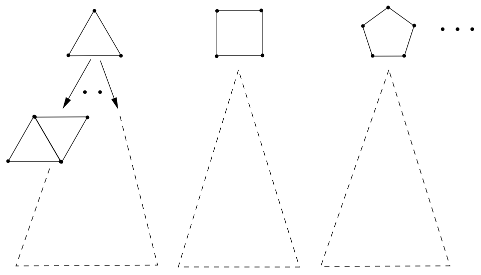
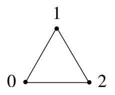


Fig. 4. Trees of generated plane graphs, with maximal face length $3, 4, 5, \ldots$

```
(let vs = [0 ..< int n];
 fs = [ Face vs Final, Face (rev vs) Nonfinal];
 b = (if n < 5 then None else Some (hd vs))
 in (Graph vs fs (int 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 values for the heights of the vertices are initially 0.

```
graph 3 =
 Graph [0, 1, 2]
      [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 Generation of the Tree of Plane Graphs

For the definition of plane graphs, we first define a generic tree function, inductively as the reachability relation induced by a given successor function *succs*. An intuitive definition of the set of graphs reachable 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 constant *Tree* :: (*graph* ⇒ *graph list*) ⇒ *graph* ⇒ *graph set* by the following two clauses:

$g \in \mathit{Tree\ succs\ g}$
$g' \in \mathit{Tree\ succs\ g} \Longrightarrow g'' \in \mathit{set\ (succs\ g')} \Longrightarrow g'' \in \mathit{Tree\ succs\ g}$

We aim at a definition of plane graphs for which we can generate executable ML code. But the ML code generated from this first definition of *Tree* does not terminate even if the defined set is finite. The reason is the depth-first evaluation strategy of inductive definitions [1]. When all elements of a finite set are enumerated, the generated function can still recursively call itself, whereas the termination condition is never reached.

For this reason we need to change the order in which one step and n steps are performed in the induction step. Then the evaluation is stopped as soon as *set* (*succs g*) is empty. This does not allow us to treat the start 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 fig. 5):

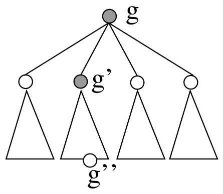


Fig. 5. Inductive definition of graph trees.

We define a constant *tree* :: (*graph* ⇒ *graph list*) ⇒ (*graph* ∗ *graph*) *set* by the following two clauses:

$(g,\ g) \in \mathit{tree\ succs}$
$g' \in \mathit{set}\ (\mathit{succs}\ g) \Longrightarrow (g',\ g'') \in \mathit{tree\ succs} \Longrightarrow (g,\ g'') \in \mathit{tree\ succs}$

Then 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.

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

The set of all terminal graphs is then the set of all graphs generated by any parameter

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

We can show that both definitions *tree* and *Tree* are equivalent.

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

3.3 The Definition of Plane Graphs

For the definition of plane graphs it is sufficient to start the 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, an integer value of the set $\{3, \ldots\}$, the size of the final face.

$\mathit{planeparameter} = \{i{::}\mathit{int}.\ 3 \leq i\}$

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*. The function *Seed param* constructs a seed graph for the parameter *param*. Using these functions , we finally define plane graphs.

Planes :: *graph set*
Planes ≡ *terminalsTree* (*Seed*::*planeparameter* ⇒ *graph*) *successorsList*

PlanesParam :: *planeparameter* ⇒ *graph set*
PlanesParam param ≡
    terminalsTreeParam param (*Seed*::*planeparameter* ⇒ *graph*) *successorsList*

PlanesTree :: *planeparameter* ⇒ *graph set*
PlanesTree param ≡ *Tree* (*successorsList param*) (*Seed param*)

This definition of plane graphs contains infinitely many trees and each tree has infinitely many elements.

4 Tame Plane Graphs

In this section we first recall the definition of tame plane graphs according to [3] and then show how they can be defined in Isabelle/HOL.

First we need to define some constants and functions: The constant 14.8 is called the *target*.

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

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

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

b(p,q)	q=0	1	2	3	4
p=0	x	x	x	7.135	10.649
1	x	x	6.95	7.135	x
2	x	8.5	4.756	12.981	x
3	x	3.642	8.334	x	x
4	4.139	3.781	x	x	x
5	0.55	11.22	x	x	x
6	6.339	x	x	x	x

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

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

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

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

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

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.
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, if

1. $\mathbf{d}(|f|) \leq w(f)$.
2. If v has type (p, q), then $\mathbf{b}(p, q) \leq \sum\limits_{v \in f} w(f)$.
3. Let V be any set of vertices of type (5,0).
   If the cardinality of V is $k \leq 4$, then $0.55k \leq \sum\limits_{V \cap f \neq \emptyset} w(f)$.
4. Let V be any separated set of vertices.
   Then $\sum\limits_{v \in V} \mathbf{a}(tri(v)) \leq \sum\limits_{V \cap f \neq \emptyset} (w(f) - d(|f|))$.

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-circuit is a face or the opposite of a face.
3. Every 4-circuit surrounds one of the cases illustrated in fig. 6.

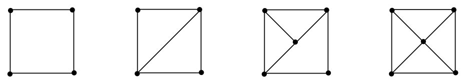


Fig. 6. Tame 4-circuits

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. $$\sum_f \mathbf{c}(|f|) \geq 8.$$
7. There exists an admissible weight assignment of total weight $\sum_f w(f)$ less than the target, 14.8.

4.1 Constants

In the following we show the implementation of these definitions in Isabelle/HOL. The names of the defined constants correspond to the numbers of properties in the definition, e.g. *tame3* corresponds to property 3 of *tame*.

The implementation is quite close to the mathematical description for the conditions of tameness that can be expressed in set theoretic formulas. Other conditions, like that 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 :: int
squanderTarget ≡ 14800

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

$squanderVertex :: int \Rightarrow int \Rightarrow int$
$\mathbf{b}\ p\ q \equiv if\ p = 0 \wedge q = 3\ then\ 7135$
$\quad else\ if\ p = 0 \wedge q = 4\ then\ 10649$
$\quad else\ if\ p = 1 \wedge q = 2\ then\ 6950$
$\quad else\ if\ p = 1 \wedge q = 3\ then\ 7135$
$\quad else\ if\ p = 2 \wedge q = 1\ then\ 8500$
$\quad else\ if\ p = 2 \wedge q = 2\ then\ 4756$
$\quad else\ if\ p = 2 \wedge q = 3\ then\ 12981$
$\quad else\ if\ p = 3 \wedge q = 1\ then\ 3642$
$\quad else\ if\ p = 3 \wedge q = 2\ then\ 8334$
$\quad else\ if\ p = 4 \wedge q = 0\ then\ 4139$
$\quad else\ if\ p = 4 \wedge q = 1\ then\ 3781$
$\quad else\ if\ p = 5 \wedge q = 0\ then\ 550$
$\quad else\ if\ p = 5 \wedge q = 1\ then\ 11220$
$\quad else\ if\ p = 6 \wedge q = 0\ then\ 6339$
$\quad else\ squanderTarget$

$scoreFace :: int \Rightarrow int$
$\mathbf{c}\ n \equiv if\ n = 3\ then\ 1000$
$\quad else\ if\ n = 4\ then\ 0$
$\quad else\ if\ n = 5\ then\ -1030$
$\quad else\ if\ n = 6\ then\ -2060$
$\quad else\ if\ n = 7\ then\ -3030$
$\quad else\ if\ n = 8\ then\ -3030$
$\quad else\ -3030$

$getSquanderFace :: int \Rightarrow int$
$\mathbf{d}\ n \equiv if\ n = 3\ then\ 0$
$\quad else\ if\ n = 4\ then\ 2378$
$\quad else\ if\ n = 5\ then\ 4896$
$\quad else\ if\ n = 6\ then\ 7414$
$\quad else\ if\ n = 7\ then\ 9932$
$\quad else\ if\ n = 8\ then\ 10916$
$\quad else\ squanderTarget$

4.2 Separation

separated1: For each vertex in V there is an exceptional face containing it:

$separated1 :: graph \Rightarrow vertex\ set \Rightarrow bool$
$separated1\ g\ V \equiv \forall v \in V.\ except\ g\ v \neq 0$

separated2: No two vertices in V are adjacent:

$separated2 :: graph \Rightarrow vertex\ set \Rightarrow bool$
$separated2\ g\ V \equiv \forall v \in V.\ \forall f \in set\ (facesAt\ g\ v).\ f \cdot v \notin V$

separated3: No two vertices lie on a common quadrilateral:

$separated3 :: graph \Rightarrow vertex\ set \Rightarrow bool$

$\mathit{separated3}\ g\ V \equiv$
$\quad \forall v \in V.\ \forall f \in \mathit{set}\ (\mathit{facesAt}\ g\ v).\ |\mathit{vertices}\ f| \leq \mathit{4} \longrightarrow \mathit{set}\ (\mathit{vertices}\ f) \cap V = \{v\}$

$\mathit{preSeparated} :: \mathit{graph} \Rightarrow \mathit{vertex\ set} \Rightarrow \mathit{bool}$
$\mathit{preSeparated}\ g\ V \equiv \mathit{separated2}\ g\ V \wedge \mathit{separated3}\ g\ V$

separated4: Every vertex in V has degree 5.

$\mathit{separated4} :: \mathit{graph} \Rightarrow \mathit{vertex\ set} \Rightarrow \mathit{bool}$
$\mathit{separated4}\ g\ V \equiv \forall v \in V.\ \mathit{degree}\ g\ v = \mathit{5}$

$\mathit{separated} :: \mathit{graph} \Rightarrow \mathit{vertex\ set} \Rightarrow \mathit{bool}$
$\mathit{separated}\ g\ V \equiv$
$\quad \mathit{separated1}\ g\ V \wedge \mathit{separated2}\ g\ V \wedge \mathit{separated3}\ g\ V \wedge \mathit{separated4}\ g\ V$

4.3 Admissibility

admissible0: A weight assignment assigns every face a non-negative value.

$\mathit{admissible0} :: (\mathit{face} \Rightarrow \mathit{int}) \Rightarrow \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{admissible0}\ w\ g \equiv \forall f \in \mathit{set}\ (\mathit{faces}\ g).\ \mathit{0} \leq w\ f$

admissible1: $\mathbf{d}(|f|) \leq w(f)$.

$\mathit{admissible1} :: (\mathit{face} \Rightarrow \mathit{int}) \Rightarrow \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{admissible1}\ w\ g \equiv \forall f \in \mathit{set}\ (\mathit{faces}\ g).\ \mathbf{d}\ |\mathit{vertices}\ f| \leq w\ f$

admissible2: If v has type (p, q), then $\mathbf{b}(p, q) \leq \sum_{v \in f} w(f)$.

$\mathit{admissible2} :: (\mathit{face} \Rightarrow \mathit{int}) \Rightarrow \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{admissible2}\ w\ g \equiv$
$\quad \forall v \in \mathit{set}\ (\mathit{vertices}\ g).\ \mathbf{b}\ (\mathit{tri}\ g\ v)\ (\mathit{quad}\ g\ v) \leq \sum_{f \in \mathit{facesAt}\ g\ v} w\ f$

admissible3: Let V be any set of vertices of type (5,0).
If the cardinality of V is $k \leq 4$, then $0.55k \leq \sum_{V \cap f \neq \emptyset} w(f)$.

$\mathit{admissible3} :: (\mathit{face} \Rightarrow \mathit{int}) \Rightarrow \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{admissible3}\ w\ g \equiv$
$\quad \forall V.\ \mathit{card}\ V \leq \mathit{4} \longrightarrow$
$\quad V \subseteq \{v.\ v \in \mathit{set}\ (\mathit{vertices}\ g) \wedge \mathit{tri}\ g\ v = \mathit{5} \wedge \mathit{quad}\ g\ v = \mathit{0}\} \longrightarrow$
$\quad \sum_{f \in [f \in \mathit{faces}\ g.\ V \cap \mathit{set}\ (\mathit{vertices}\ f) \neq \{\}]} w\ f \leq \mathit{550} * \mathit{int}\ (\mathit{card}\ V)$

admissible4: Let V be any separated set of vertices.
Then $\sum_{v \in V} \mathbf{a}(tri(v)) \leq \sum_{V \cap f \neq \emptyset} (w(f) - d(|f|))$.

$admissible4 :: (face \Rightarrow int) \Rightarrow graph \Rightarrow bool$
$admissible4\ w\ g \equiv$
$\quad \forall V.\ separated\ g\ (set\ V) \longrightarrow$
$\quad set\ V \subseteq set\ (vertices\ g) \longrightarrow$
$\qquad (\sum_{v \in V} \mathbf{a}\ (tri\ g\ v))$
$\quad + (\sum_{f \in [f \in faces\ g.\ \exists v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} \mathbf{d}\ |vertices\ f|\ )$
$\quad \leq \sum_{f \in [f \in faces\ g.\ \exists v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} w\ f$

$admissible :: (face \Rightarrow int) \Rightarrow graph \Rightarrow bool$
$admissible\ w\ g \equiv$
$\qquad admissible0\ w\ g \wedge admissible1\ w\ g \wedge admissible2\ w\ g \wedge admissible3\ w\ g$
$\qquad \wedge\ admissible4\ w\ g$

4.4 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 fig. 7). During the verification

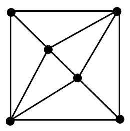


Fig. 7. Two adjacent vertices of type $(4, 0)$

it turned out that the original definition of tame graphs by Hales in fact allows 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-circuit. On the outside one of the tame configurations of fig. 6, discarding any that give fewer than 8 triangles.

Hales [private communication] suggested that we strengthen the notion of tameness to match the algorithm because it can be shown that all counterexamples must satisfy the stronger notion. Therefore we extend the definition of *tame* by a new restriction *tame8* that no two adjacent vertices of type $(4, 0)$ occur in a tame graph. Properties *tame1* to *tame7* correspond to properties 1 to 7 of the original definition.

tame1: The length of each face is (at least 3 and) at most 8:

$tame1 :: graph \Rightarrow bool$
$tame1\ g \equiv \forall f \in set\ (faces\ g).\ 3 \leq |vertices\ f| \wedge |vertices\ f| \leq 8$

tame2: Every 3-circuit is a face or the opposite of a face:

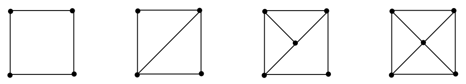
A face given by a vertex list *vs* is contained in a graph *g*, if it is isomorphic to one of the faces in *g*. The notation $f \in_{\cong} F$ means $\exists f' \in F.\ f \cong f'$, where $\cong$ is the equivalence relation on faces (see §6).

A 3-circuit in a graph *g* is a path of length 3 along any faces of *g*.

```
triangle :: vertex ⇒ vertex ⇒ vertex ⇒ graph  ⇒ bool
triangle a b c g ≡
    (∃f ∈ set (faces g). (a, b) ∈ edges f)
  ∧ (∃f ∈ set (faces g). (b, c) ∈ edges f)
  ∧ (∃f ∈ set (faces g). (c, a) ∈ edges f)

tame2 :: graph ⇒ bool
tame2 g ≡
   ∀ a b c. triangle a b c g ⟶
   (Face [a, b, c] Final) ∈≅ set (faces g) ∨
   (Face [c, b, a] Final)∈≅ set (faces g)
```

tame3 : Every 4-circuit surrounds one of the following configurations:



A 4-circuit in a graph *g* is a path of length 4 along any faces of *g*.

```
quadrilateral :: vertex ⇒ vertex ⇒ vertex ⇒ vertex ⇒ graph ⇒ bool
quadrilateral a b c d g ≡
    (∃f ∈ set (faces g). (a, b) ∈ edges f)
  ∧ (∃f ∈ set (faces g). (b, c) ∈ edges f)
  ∧ (∃f ∈ set (faces g). (c, d) ∈ edges f)
  ∧ (∃f ∈ set (faces g). (d, a) ∈ edges f)

tameConf1 :: vertex ⇒ vertex ⇒ vertex ⇒ vertex ⇒ face set
tameConf1 a b c d ≡ {Face [a, b, c, d] Final}

tameConf2 :: vertex ⇒ vertex ⇒ vertex ⇒ vertex ⇒ face set
tameConf2 a b c d ≡ {Face [a, b, c] Final, Face [a, c, d] Final}

tameConf3 :: vertex ⇒ vertex ⇒ vertex ⇒ vertex ⇒ vertex ⇒ face set
tameConf3 a b c d e ≡
   {Face [a, b, e] Final, Face [b, c, e] Final, Face [a, e, c, d] Final}

tameConf4 :: vertex ⇒ vertex ⇒ vertex ⇒ vertex ⇒ vertex ⇒ face set
tameConf4 a b c d e ≡
   {Face [a, b, e] Final, Face [b, c, e] Final, Face [c, d, e] Final,
   Face [d, a , e] Final}
```

Given a fixed 4-circuit, and using the convention of drawing faces clockwise, a tame configuration can occur in the 'interior' or on the outside of the 4-circuit. For configuration 2 there are two possible arrangements of the triangles, for configuration 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 assures 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.

$\mathit{tame4circuit} :: \mathit{graph} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{vertex} \Rightarrow \mathit{bool}$
$\mathit{tame4circuit}\ g\ a\ b\ c\ d \equiv$
$\quad \exists e.\ \mathit{tameConf1}\ a\ b\ c\ d \subseteq_{\cong} \mathit{set}\ (\mathit{faces}\ g)$
$\quad\ \ \lor \mathit{tameConf2}\ a\ b\ c\ d \subseteq_{\cong} \mathit{set}\ (\mathit{faces}\ g)$
$\quad\ \ \lor \mathit{tameConf2}\ b\ c\ d\ a \subseteq_{\cong} \mathit{set}\ (\mathit{faces}\ g)$
$\quad\ \ \lor \mathit{tameConf3}\ a\ b\ c\ d\ e \subseteq_{\cong} \mathit{set}\ (\mathit{faces}\ g)$
$\quad\ \ \lor \mathit{tameConf3}\ b\ c\ d\ a\ e \subseteq_{\cong} \mathit{set}\ (\mathit{faces}\ g)$
$\quad\ \ \lor \mathit{tameConf3}\ c\ d\ a\ b\ e \subseteq_{\cong} \mathit{set}\ (\mathit{faces}\ g)$
$\quad\ \ \lor \mathit{tameConf3}\ d\ a\ b\ c\ e \subseteq_{\cong} \mathit{set}\ (\mathit{faces}\ g)$
$\quad\ \ \lor \mathit{tameConf4}\ a\ b\ c\ d\ e \subseteq_{\cong} \mathit{set}\ (\mathit{faces}\ g)$

$\mathit{tame3} :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame3}\ g \equiv \forall a\ b\ c\ d.\ \mathit{quadrilateral}\ a\ b\ c\ d\ g \longrightarrow$
$\quad \mathit{tame4circuit}\ g\ a\ b\ c\ d \lor \mathit{tame4circuit}\ g\ d\ c\ b\ a$

tame4: The degree of every vertex is (at least 2 and) at most 6:

$\mathit{tame4} :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame4}\ g \equiv \forall v \in \mathit{set}\ (\mathit{vertices}\ g).\ 2 \le \mathit{degree}\ g\ v \land \mathit{degree}\ g\ v \le 6$

tame5: If a vertex is contained in an exceptional face, then the degree of the vertex is at most 5:

$\mathit{tame5} :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame5}\ g \equiv$
$\quad \forall f \in \mathit{set}\ (\mathit{faces}\ g).\ \forall v \in \mathit{set}\ (\mathit{vertices}\ f).\ 5 \le |\mathit{vertices}\ f| \longrightarrow \mathit{degree}\ g\ v \le 5$

tame6: $8 \le \sum_f c(|f|)$:

$\mathit{tame6} :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame6}\ g \equiv 8000 \le \sum_{f \in \mathit{faces}\ g} \mathbf{c}\ |\mathit{vertices}\ f|$

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

tame7: There exists an admissible weight assignment of total weight less than the target:

$\mathit{tame7} :: \mathit{graph} \Rightarrow \mathit{bool}$
$\mathit{tame7}\ g \equiv \exists w.\ \mathit{admissible}\ w\ g \land \sum_{f \in \mathit{faces}\ g} w\ f < \mathit{squanderTarget}$

Property *tame7* assures that the set of tame plane graphs is finite.

tame8: We formalize the additional restriction (compared with the original definition) that tame graphs do not contain two adjacent vertices of type $(4, 0)$.

```
$vertexHas40$ :: $graph \Rightarrow vertex \Rightarrow bool$
$vertexHas40\ g\ v \equiv$
    $finalVertex\ g\ v \wedge tri\ g\ v = 4 \wedge quad\ g\ v = 0 \wedge except\ g\ v = 0$

$neighbours$ :: $graph \Rightarrow vertex \Rightarrow vertex\ list$
$neighbours\ g\ v \equiv map\ (\lambda f.\ f \cdot v)\ (facesAt\ g\ v)$

$hasAdjacent40$ :: $graph \Rightarrow bool$
$hasAdjacent40\ g \equiv$
    $\exists\, v \in set\ (vertices\ g).\ vertexHas40\ g\ v\ \wedge$
    $(\exists\, w \in set\ (neighbours\ g\ v).\ vertexHas40\ g\ w$
      $\wedge\ remlist\ (neighbours\ g\ v)\ (w\#(neighbours\ g\ w)) \neq [])$

$tame8$ :: $graph \Rightarrow bool$
$tame8\ g \equiv \neg\ hasAdjacent40\ g$

$tame$ :: $graph \Rightarrow bool$
$tame\ g \equiv$
    $tame1\ g \wedge tame2\ g \wedge tame3\ g \wedge tame4\ g \wedge tame5\ g \wedge tame6\ g \wedge tame7\ g$
  $\wedge\ tame8\ g$
```

5 Refinements

Starting from the definition we obtain a first algorithm to enumerate all tame plane graphs: Enumerate all plane graphs and remove all graphs that are demonstrably not tame. It is not necessary to remove all graphs that are not tame, its sufficient to generate a set of plane graphs that contains all tame plane graphs. However, this first algorithm is not terminating, since the set of all plane graphs is not finite.

To overcome this problem, we use the following approach: We start with the set of all plane graphs ($\text{Ref}_0 = Planes$ in fig.8). We gradually reduce the generated set of graphs, imposing the restrictions of tameness to it ($\text{Ref}_1, \ldots, \text{Ref}_4$), 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) ($\text{Ref}_5 = enum$).

There are two different reasons why a graph g can be neglected:

(I) if all final graphs generated by g are not tame.
(II) if for every final graph generated by g an isomorphic graph will be generated by another path in the tree.

5.1 Ref_1: Fixed face and edge

In the definition of plane graphs successors for all edges in all nonfinal faces are calculated. The first refinement step is to fix one nonfinal face and one

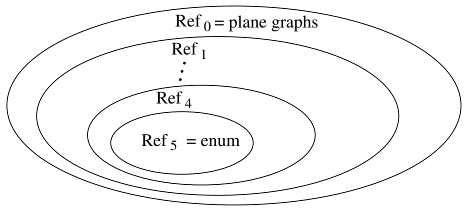


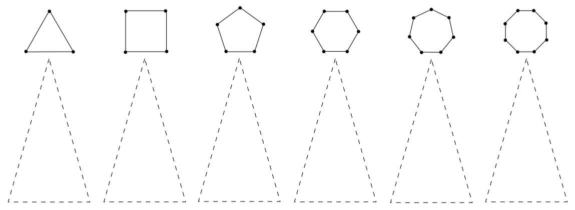
Fig. 8. Proof structure

edge in this face, where the possible successors are calculated. We denote the generated set of graphs by $Ref_1{}'$. This does not reduce the set of generated graphs modulo isomorphism, only some graphs are left out, when isomorphic graphs are generated by another path in the tree. (This is an optimization of type (I).) Hence this is still 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 it does not matter which nonfinal face we treat first.

For the verification it does not matter which of the faces and which of the vertices we select.

5.2 Ref$_2$: Restriction to graphs with maximum face size 8

Since all tame graphs have maximum face sizes 8, 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 not be tame. This is an optimization of type (II).



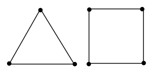
We restrict the set of seed parameters to the finite set $\{3, \ldots, 8\}$.

5.3 Ref$_3$: Complex Seed Graphs

As the next refinement step we replace the first two seed graphs (consisting of a final triangle, a final quadrilateral, resp.) by a new finite set of complex seed graphs (see fig.9). Every new seed graph has one final vertex v, and it consists of t final triangles and q final quadrilaterals all incident with v and one nonfinal face.

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* $\{5, \dots, 8\}$.

In the second step we replace the two quad seed graphs



by an (infinite) set of complex seed graphs (*vertex seed graphs*) with one final vertex v (of degree at least 2), each seed graph is represented by a *vertex parameter*, a list of quad parameters (of the length of the degree of v), the sizes of the faces incident with v in cyclic order.

From the set of vertex seed graphs every seed graph with $14.8 \leq \mathbf{b}(p, q)$ can be excluded, since every graph generated from these graphs will not be tame: it contains a vertex v with $14.8 \leq \mathbf{b}(p, q)$ and for every admissible weight assignment w, $14.8 \leq \mathbf{b}(p, q) \leq \sum_{f \in facesg} wf$. This violates properties (*tame7*) and (*admissible2*).

Hence we make a list of all types (p, q) with $14.8 < \mathbf{b}(p, q)$ and create all seed graphs (up to isomorphism) of this type. This results in a list of 17 seed graphs shown in fig. 9. We impose a fixed order on these seed graphs by assigning each graph an index out of the set $\{0, \dots, 16\}$. Every graph is represented by this index as parameter.

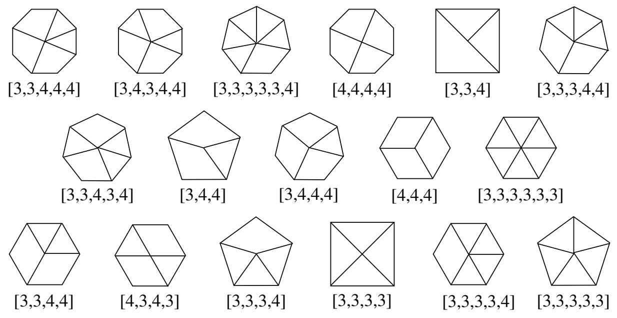


Fig. 9. Seed for quad parameter

We can neglect all plane graphs g generated by a seed s, that contain a vertex of same type as the final vertex in one earlier seed graph s'. The graph g has already been generated by another path starting from this earlier seed graph (see fig. 10)

This refinements avoids generation of isomorphic copies of tame plane graphs and improves the efficiency of the generation.

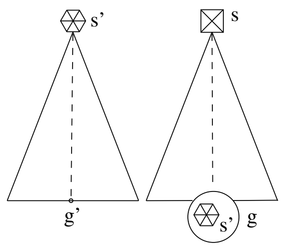


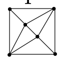
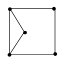
Fig. 10. neglect graphs that contain earlier seed graphs

5.4 Ref$_4$: Neglectable nonfinal graphs

We can neglect nonfinal graphs by the following refinements:

- The successor function can be simplified such that it replaces all nonfinal triangles by final ones. (property *tame2*)
- The successor function can be further simplified. If a graph contains a nonfinal quad, it can be replaced by all configurations that are allowed by property *tame3*.
- A new generated (nonfinal) graph can be neglected if an enclosed vertex is created. This would contradict property *tame2*.
- We calculate a lower bound for $\sum_f w(f)$ for a nonfinal graph. A graph can be neglected if a lower bound of $\sum_f w(f)$ exceeds 14.8.

5.5 Ref$_5$ = *enum*: Neglectable Final Graphs

- We calculate a lower bound for $\sum_f w(f)$ for a nonfinal graph. A graph can be neglected if a lower bound of $\sum_f w(f)$ exceeds 14.8.
- A graph can be neglected if $\sum_f c(|f|) < 8$ (*tame6*).
- A graph can be neglected if it contains vertices whose degree is too high (*tame4*, *tame5*).
- A graph can be neglected if it contains some forbidden configurations, like  (*tame8*) or  (*tame3*, *tame6*).

So far, this part is formally proved in Isabelle.

6 Plane Graph Isomorphism

Plane graph isomorphism has already been defined informally in §2 and is referred to in justifying a number of the refinement steps in §5. This section provides both a formal definition and an executable implementation of plane graph isomorphism, and it uses the implementation to remove redundancies from Hales' archive of tame plane graphs (see §1).

We work with two simple representations of plane graphs:

$'a\ Fgraph = {'a}\ list\ set$
$'a\ fsgraph = {'a}\ list\ list$

A face is a list of nodes (of type $'a$), and plane graph is either a set (*Fgraph*) or a list (*fsgraph*) of faces: lists are used instead of sets whenever we need executability.

Two faces are considered equivalent if one can be obtained from the other by rotation. This is formalized as the equivalence relation *EqF* on faces. Note that $A\ //\ R$ is the quotient of a set A by an equivalence R.

The notion of a *proper homomorphism* and *isomorphism* is defined both on the set and list representation of plane graphs.

$is\text{-}pr\text{-}Hom :: ('a \Rightarrow 'b) \Rightarrow 'a\ Fgraph \Rightarrow 'b\ Fgraph \Rightarrow bool$
$is\text{-}pr\text{-}Hom\ \varphi\ Fs_1\ Fs_2\ \equiv\ (map\ \varphi\ `\ Fs_1)//EqF = Fs_2\ //\ EqF$

$is\text{-}pr\text{-}Iso :: ('a \Rightarrow 'b) \Rightarrow 'a\ Fgraph \Rightarrow 'b\ Fgraph \Rightarrow bool$
$is\text{-}pr\text{-}Iso\ \varphi\ Fs_1\ Fs_2\ \equiv\ is\text{-}pr\text{-}Hom\ \varphi\ Fs_1\ Fs_2 \wedge inj\text{-}on\ \varphi\ (\bigcup F \in Fs_1.\ set\ F)$

$is\text{-}pr\text{-}hom :: ('a \Rightarrow 'b) \Rightarrow 'a\ fsgraph \Rightarrow 'b\ fsgraph \Rightarrow bool$
$is\text{-}pr\text{-}hom\ \varphi\ Fs_1\ Fs_2\ \equiv\ is\text{-}pr\text{-}Hom\ \varphi\ (set\ Fs_1)\ (set\ Fs_2)$

$is\text{-}pr\text{-}iso :: ('a \Rightarrow 'b) \Rightarrow 'a\ fsgraph \Rightarrow 'b\ fsgraph \Rightarrow bool$
$is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2\ \equiv\ is\text{-}pr\text{-}Iso\ \varphi\ (set\ Fs_1)\ (set\ Fs_2)$

6.1 An Executable Isomorphism Test

In a stepwise development (which we cannot detail here) we arrive at an executable isomorphism test based on a representation of morphisms as lists of pairs. Function *test* checks if two morphisms are compatible:

$test :: ('a \times 'b)list \Rightarrow ('a \times 'b)list \Rightarrow bool$
$test\ I\ I'\ \equiv$
$\quad list\text{-}all\ (\lambda xy.\ list\text{-}all\ (\lambda xy'.\ (fst\ xy = fst\ xy') = (snd\ xy = snd\ xy'))\ I')\ I$

and *merge* merges two compatible morphisms:

$merge :: ('a \times 'b)list \Rightarrow ('a \times 'b)list \Rightarrow ('a \times 'b)list$
$merge\ []\ I = I$
$merge\ (xy\#xys)\ I = (let\ (x,y) = xy\ in$
$\quad if\ list\text{-}all\ (\lambda(x',y').\ x \neq x')\ I\ then\ xy\ \#\ merge\ xys\ I$
$\quad else\ merge\ xys\ I)$

Note that *fst*/*snd* is the first/second component of a pair and function *list-all*/*list-ex* checks if all/some element of a list satisfies a given test.

The actual isomorphism test tries to pair faces of the same length and iterates over all rotations of one of the two faces. If the current isomorphism I can be extended with I' (the result of pairing the two new faces), then the search continues, otherwise it fails:

```
$iso\text{-}test :: ('a \times 'b)list \Rightarrow 'a\ fsgraph \Rightarrow 'b\ fsgraph \Rightarrow bool$
$iso\text{-}test\ I\ []\ Fs_2 = (Fs_2 = [])$
$iso\text{-}test\ I\ (F_1\#Fs_1)\ Fs_2 =$
  $list\text{-}ex\ (\lambda F_2.\ length\ F_1 = length\ F_2 \wedge$
    $list\text{-}ex\ (\lambda n.\ let\ I' = zip\ F_1\ (rotate\ n\ F_2)\ in$
      $if\ \ test\ I'\ I\ \ then\ iso\text{-}test\ (merge\ I'\ I)\ Fs_1\ (remove1\ F_2\ Fs_2)\ else\ False)$
     $[0\ ..<\ length\ F_2])\ Fs_2$
```

The correctness theorem is littered with many preconditions which simply express that two representations are indeed those of proper plane graphs, e.g. all nodes in a face are distinct:

$$\begin{array}{l} [\![\ \forall F{\in}set\ Fs_1.\ distinct\ F;\ \forall F{\in}set\ Fs_2.\ distinct\ F;\ [] \notin set\ Fs_2; \\ \quad distinct\ Fs_1;\ inj\text{-}on\ (\lambda xs.\{xs\}//EqF)\ (set\ Fs_1); \\ \quad distinct\ Fs_2;\ inj\text{-}on\ (\lambda xs.\{xs\}//EqF)\ (set\ Fs_2)\ ]\!] \Longrightarrow \\ \qquad iso\text{-}test\ []\ Fs_1\ Fs_2 = (\exists \varphi.\ is\text{-}pr\text{-}iso\ \varphi\ Fs_1\ Fs_2) \end{array}$$

To obtain acceptable performance, we need to test right away if the two graphs have the same number of faces (which is easy) and nodes. For the latter test we pair each graph with the number of its nodes in order not to have to recompute this all the time:

$$\begin{array}{l} iso\text{-}test2 :: (nat \times 'a\ fsgraph) \Rightarrow (nat \times 'b\ fsgraph) \Rightarrow bool \\ iso\text{-}test2\ (n_1,Fs_1)\ (n_2,Fs_2) \equiv \\ \quad n_1 = n_2 \wedge length\ Fs_1 = length\ Fs_2 \wedge iso\text{-}test\ []\ Fs_1\ Fs_2 \end{array}$$

Finally we move from *proper* isomorphisms to isomorphisms where by allowing the orientation of all faces in one of the graphs to be reversed:

$$\begin{array}{l} is\text{-}Iso :: ('a \Rightarrow 'b) \Rightarrow 'a\ Fgraph \Rightarrow 'b\ Fgraph \Rightarrow bool \\ is\text{-}Iso\ \varphi\ Fs_1\ Fs_2 \equiv is\text{-}pr\text{-}Iso\ \varphi\ Fs_1\ Fs_2 \vee is\text{-}pr\text{-}Iso\ \varphi\ Fs_1\ (rev\ `\ Fs_2) \end{array}$$

$$\begin{array}{l} is\text{-}iso :: ('a \Rightarrow 'b) \Rightarrow 'a\ fsgraph \Rightarrow 'b\ fsgraph \Rightarrow bool \\ is\text{-}iso\ \varphi\ Fs_1\ Fs_2 \equiv is\text{-}Iso\ \varphi\ (set\ Fs_1)\ (set\ Fs_2) \end{array}$$

where f ‘ A is the image of a set A under a function f. The executable version is obvious

$$\begin{array}{l} iso :: (nat \times 'a\ fsgraph) \Rightarrow (nat \times 'b\ fsgraph) \Rightarrow bool \\ iso\ g_1\ g_2\ \equiv\ iso\text{-}test2\ g_1\ g_2 \vee iso\text{-}test2\ g_1\ (fst\ g_2,\ map\ rev\ (snd\ g_2)) \end{array}$$

and its correctness theorem is very similar to the one for *iso-test*.

6.2 A Reduced Archive

Hales published both his Java programs that enumerate all tame plane graphs and a set of files containing all tame plane graphs [2]. The latter files are referred to as the *archive* and are found in subdirectory `JavaKep02/graph00/src/graph/archive/`. Upon examining the archive we found that it contains not just the enumerated tame plane graphs but also junk, i.e. isomorphic copies of graphs or graphs that never show up in the enumeration.

We have exported function *iso* to ML [1] and have used it filter out those graphs in the archive that are not isomorphic to one in the enumeration *enum*. It turns out that of the 5128 graphs in the archive, only 2872 remain, the rest are redundant. A reduced archive is now found here [4].

Note that Hales never claims that his archive is free of junk. The advantage of our reduced archive is that it reduces the computation time (and potentially also the cleverness) of subsequent proof steps.

7 Conclusion

Now, collecting all completeness theorems of §5, we can finally prove Theorem 1:

Theorem $g \in \mathit{Planes} \Longrightarrow \mathit{tame}\ g \Longrightarrow \exists\, h \in \mathit{Ref}_1.\ g \cong h$
Theorem $g \in \mathit{Ref}_1 \Longrightarrow \mathit{tame}\ g \Longrightarrow \exists\, h \in \mathit{Ref}_2.\ g \cong h$

$\vdots$

Theorem $g \in \mathit{Ref}_4 \Longrightarrow \mathit{tame}\ g \Longrightarrow \exists\, h \in \mathit{enum}.\ g \cong h$

we finally obtain the result:

Theorem $g \in \mathit{Planes} \Longrightarrow \mathit{tame}\ g \Longrightarrow \exists\, h \in \mathit{enum}.\ g \cong h$

Generating ML code from the definition of *enum*, executing it and checking that for every generated graphs there is an isomorphic graph in the archive yields a confirmation of Theorem 1.

References

1. 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.
2. T. Hales. The Kepler Conjecture, 2002. `http://www.math.pitt.edu/~thales/kepler02/javakep02.tar`.
3. T. Hales. A Proof of the Kepler Conjecture. `http://www.math.pitt.edu/~thales/kepler04/fullkepler.pdf`, 2004.
4. T. Nipkow. Reduced Archive of Tame Plane Graphs, 2005. `http://www.in.tum.de/~nipkow/Flyspeck/`.
5. 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/`.