Flyspeck I: Tame Graphs

Tobias Nipkow and Gertrud Bauer and Paula Schultz†

Institut für Informatik, TU München

Abstract. We present a verified enumeration of tame graphs as defined in Hales' proof of the Kepler Conjecture and confirm the completeness of Hales' list of all tame graphs while reducing it from 5128 to 2771 graphs.

1 Introduction

In 1611 Kepler asserted what every cannoneer of the day must have known, that the so-called cannonball packing is a densest arrangement of 3-dimensional balls of the same size. In 1900 this assertion became part of Hilbert's 18th problem. In 1998 Thomas Hales announced the first (by now) accepted proof. It involves 3 distinct large computations. After 4 years of refereeing by a team of 12 referees, an abridged version was published only recently [5]. The referees declared that they were 99% certain of the correctness of the proof. Dissatisfied with this state of affairs Hales started the informal open-to-all collaborative *flyspeck* project (www.math.pitt.edu/~thales/flyspeck) to formalize the whole proof with a theorem prover. This paper is the first definite contribution to flyspeck.

Hales' proof goes roughly like this: any potential counter example (denser packing) gives rise to a "tame" plane graph, where tameness is a very specific notion; enumerate all (finitely many) tame graphs (by computer); for each of them check (again by computer) that they cannot constitute a counter example. For modularity reasons Hales provided the *Archive*, a collection of files with (hopefully) all tame graphs.

We recast Hales' Java program for the enumeration of all tame graphs in logic (Isabelle/HOL), proved its completeness, ran it, and compared the output to Hales' Archive. It turns out that Hales was right, the Archive is complete, although redundant (there are at most 2771, not 5128 tame graphs), and that one tameness condition present in his Java program was missing from his proof text. Apart from the contribution to Hales' proof, this paper demonstrates that theorem provers can not just formalize known results but can help in establishing the validity of emerging state-of-the-art mathematical proofs.

An intrinsic feature of this proof, which it shares with Gonthier's proof of the Four Colour Theorem [3], is the need to perform massive computations involving the defined functions (§1.3, §5). Hence efficiency is a concern: during the development phase it is not very productive if, after every change, it takes a week to rerun the proof to find that the change broke it. Part of the achievement of our work is narrowing the gap between the specification of tameness and the enumeration (to simplify the proof) without compromising efficiency unduly.

Here is one motivating glimpse of where the tame graphs come from: each one is the result of taking a cluster of balls and projecting the centers of the balls on to the surface of the ball in the center, connecting two centers if they are within a certain distance. For an eminently readable overview of Hales' proof see [4], for the details see [5], and for the full monty read [6] and accompanying articles in the same volume. For Hales' Java program and his Archive see his web page or the online material accompanying [5]. The gory details of our own work can be found in the Isabelle theories available from the first author's home page. The thesis [1] also contains full details but is a precursor to the work presented here: the enumeration and the proof have changed considerably and the proof has been completed.

1.1 Related Work

Obua [10] and Zumkeller [11] work towards the verification of the remaining computational parts in Hales' proof.

Gonthier's proof of the Four Colour Theorem [3] is very much concerned with efficient data structures for various computations on plane graphs, a feature it shares with our proof. At the same time he employs *hypermaps* as a unifying representation of plane graphs. Potentially, hypermaps could play the same role in the less computational but mathematically more abstract parts of flyspeck.

1.2 Basic Notation

Isabelle/HOL [9] conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

Types The basic types of truth values, natural numbers and integers are called *bool*, *nat*, and *int*. The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t{::}\tau$ means that term t has type τ.

Sets (type $'a\ set$) come with their usual syntax. The pointwise image of set M under a function f is written $f\ `\ M$.

Lists (type $'a\ list$) come with the empty list [], the infix constructor $\cdot$, the infix @ that appends two lists, and the conversion function *set* from lists to sets. Function *hd* yields the head of a list, *last* the last element, and *butlast* drops the last element. Variable names ending in "s" usually stand for lists, $|xs|$ is the length of *xs*, *distinct xs* means that the elements of *xs* are all distinct. Instead of *map f xs* and *filter P xs* we frequently write $[f\ x.\ x \in xs]$ and $[x{\in}xs\ .\ P\ x]$.

1.3 Proof by Evaluation

Many theorem provers (ACL2, Coq and PVS) have the ability to evaluate functions during proof by compilation into some efficient format followed by execution. Isabelle has been able to generate ML code for some time now [2]. Recently this has been made available as an inference rule: given a term t, all functions in t are compiled to ML (provided this is possible), t is reduced to u by the ML system, and the result is turned into the theorem $t = u$. Essentially, the ML system is used as an efficient term rewriting engine. To speed things up further, *nat* and *int* are implemented by arbitrary precision ML integers.

In order to remove constructs which are not by themselves executable, code generation is preceded by a preprocessor that rewrites the term with specified lemmas. For example, the lemmas $\forall x \in set\ xs.\ P\ x \equiv list\text{-}all\ P\ xs$ and $\exists x \in set\ xs.\ P\ x \equiv list\text{-}ex\ P\ xs$ replace bounded quantifiers over lists with executable functions. This is the key to bridging the gap between specification and implementation automatically and safely.

Because lists are executable but sets are not, we sometimes phrase concepts in terms of lists rather than sets to avoid having to define the concept twice and having to provide a trivial implementation proof.

2 Plane Graphs

Following Hales we represent finite, undirected, plane graphs as lists (= finite sets) of faces and faces as lists of vertices. Note that by representing faces as lists they have an orientation. Our enumeration of plane graphs requires an additional distinction between *final* and *nonfinal* faces. This flag is attached directly to each face:

$$vertex = nat, \quad face = vertex\ list \times bool, \quad graph = face\ list$$

The projection functions for faces are called *vertices* and *final*. The *size* of a face is the length of its vertex list. Occasionally we call a list of vertices a face, too. A graph is final if all its faces are final. Function $\mathcal{F}$ returns the set of faces of a graph, i.e. is a synonym for function *set*. Function $\mathcal{V}$ returns the set of vertices in a graph, *countVertices* the number of vertices. Given a graph g and a vertex v, *facesAt* g v computes the list of faces incident to v.

For navigation around a face f we consider its list of vertices as cyclic and introduce the following notation: if v is a vertex in f then $f \cdot v$ is the vertex following v and $f^i \cdot v$ is the ith vertex following v (where i may also be -1). This description is ambiguous if there are multiple occurrences of v but this cannot arise in our context.

Representing faces as lists means that we want to regard two vertex lists *us* and *vs* as equivalent if one can be obtained from the other by rotation, in which case we write $us \cong vs$. We introduce the notation

$$x \in_{\cong} M \equiv \exists y \in M.\ x \cong y, \qquad M \subseteq_{\cong} N \equiv \forall x \in M.\ x \in_{\cong} N$$

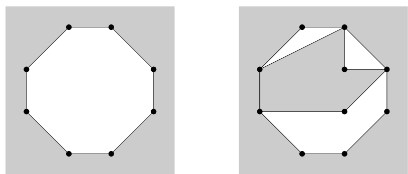
Throughout most of this paper we pretend that a graph is just a face list, but in reality it is more complicated. To avoid recomputation, *countVertices* and *facesAt* are (hidden) components of the graph.

2.1 Enumeration of Plane Graphs

Not every list of faces constitutes a plane graph. Hence we need additional means of characterizing planarity. We have chosen an operational characterization, i.e.

an executable enumeration due to Hales. The reason is that we can then view the enumeration of tame graphs as a restriction of the enumeration of plane graphs. The justification for not starting with a more abstract traditional characterization of planarity is that this is the first definite contribution to flyspeck and it is not yet clear which notion of planarity is most suitable for the rest of the project. In a nutshell, we wanted to concentrate on the new (and unchecked by referees!) enumeration of tame graphs rather than the mathematically well understood issue of planarity.

The graphs required for Hales' proof are plane graphs with at least 2 faces (including the outer one), where each face is a simple polygon of size ≥ 3. In the sequel the term *plane* refers to this class. Hales' enumeration of plane graphs proceeds inductively: you start with a seed graph with two faces, the final outer one and the (reverse) nonfinal inner one. If a graph contains a nonfinal face, it can be subdivided into a final face and any number of nonfinal ones as shown below. Final faces are grey, nonfinal ones white. The unbounded grey square indicates the unbounded outer face.



Because a face can be subdivided in many ways, this process defines a tree of graphs. By construction the leaves must be final graphs, and they are the plane graphs we are interested in: any plane graph (in the above sense) of n faces can be generated in $n-1$ steps by this process, adding one (final) face at a time.

This definition is also meant to serve as the basis of the actual enumeration. Hence we reduce its redundancy, i.e. the number of times each graph is generated, by the following means:

- The enumeration is parameterized by a natural number p which controls the maximal size of final faces in the generated graphs. The seed graph contains two $(p+3)$-gons and the final face created in each step may at most be a $(p+3)$-gon. As a result, different parameters lead to disjoint sets of graphs. Note that the nonfinal faces may (and need to be) of arbitrary size.
- In each step we subdivide only one fixed face and the new final face always shares one fixed edge with the subdivided face; which face and edge are chosen is immaterial. This does not affect the set of final graphs that can be generated but merely the order in which the final faces are created.

Formalization Now we are ready for the top level formal specification:

$\mathit{PlaneGraphs} \equiv \bigcup p\ \{g \mid \mathit{Seed}_p\ [\mathit{next\text{-}plane}_p]\rightarrow^*\ g \wedge \mathit{final}\ g\}$

where $\mathit{Seed}_p \equiv [([0,\ldots,p+2], \mathit{True}),\ ([p+2,\ldots,0], \mathit{False})]$ is the seed graph described above. Notation $g_0\ [f]\rightarrow^*\ g$ is simply suggestive syntax for $(g_0,\ g) \in$

$\{(g,\ g') \mid g' \in set\ (f\ g)\}^*$, i.e. we can reach g from g_0 in finitely many steps via function f :: *graph* $\Rightarrow$ *graph list*. In our case f is $next\text{-}plane_p$ which maps a graph to a list of successor graphs:

$next\text{-}plane_p\ g \equiv$
let $fs = [f{\in}faces\ g\ .\ \neg\ final\ f]$
in if $fs = []$ *then* $[]$
  else let $f = minimalFace\ fs;\ v = minimalVertex\ g\ f$
    in $\bigsqcup_{i\in\ [3..p\ +\ 3]}\ generatePolygon\ i\ v\ f\ g$

If there are only final faces, we are done. Otherwise we pick a minimal nonfinal face (in terms of size) and a minimal vertex within that face. Minimality of the vertex refers to its distance from the vertices in the seed graph. This policy favours compact graphs over stringy objects. Its implementation requires an additional (hidden) component in each graph. But since the choice of vertex (and face) is irrelevant as far as completeness of the enumeration is concerned, so is the precise implementation.

Having fixed f and v we subdivide f in all possible ways by placing an i-gon inside it (along the edge from v to its predecessor vertex in f), where i ranges from 3 to $p{+}3$. Function *generatePolygon* returns a list of all possible successor graphs and the suggestive syntax $\bigsqcup_{i\in\ I}\ F\ i$ represents the concatenation of all $F\ i$ for i in I.

Function *generatePolygon* operates and is explained in stages:

$generatePolygon\ n\ v\ f\ g \equiv$
let $enumeration = enumerator\ n\ |vertices\ f|;$
  $enumeration = [is{\in}enumeration\ .\ \neg\ containsDuplicateEdge\ g\ f\ v\ is];$
  $vertexLists = [indexToVertexList\ f\ v\ is.\ is \in enumeration]$
in $[subdivFace\ g\ f\ vs.\ vs \in vertexLists]$

Enumeration We have to enumerate all possible ways of inscribing a final n-gon inside f such that it shares the edge $(f^{-1} \cdot v,\ v)$ with f (which is removed). The new n-gon can in fact share all edges with f, in which case we simply finalize f without adding any new nonfinal faces; or it can touch f only at $f^{-1} \cdot v$ and v and all of its other vertices are new; or anything in between. Following Hales one can describe each of these n-gons by a list of length n of increasing indices from the interval $\{0,\ldots,|vertices\ f| - 1\}$. Roughly speaking, index i represents vertex $f^i \cdot v$ and a pair i,j of adjacent list elements is interpreted as follows: if $i < j$ then the new polygon contains an edge from vertex $f^i \cdot v$ to $f^j \cdot v$; if $i = j$ then the new polygon contains a new vertex at that point. For example, given the face $[v_0,\ldots,v_5]$, the index list $[0,2,3,3,3,5]$ represents some face $[v_0,v_2,v_3,x,y,v_5]$ where x and y are new vertices.

The enumeration of all these index lists is the task of function *enumerator* which returns a *nat list list*. We have proved that *enumerator* $n\ m$ returns all (not necessarily strictly) increasing lists of length n starting with 0 and ending with $m - 1$:

$set\ (enumerator\ n\ m) =$

$\{is \mid |is| = n \wedge hd\ is = 0 \wedge last\ is = m - 1 \wedge last\ (butlast\ is) < last\ is \wedge$
$\quad increasing\ is\}$

Condition $last\ (butlast\ is) < last\ is$ excludes lists like $[\ldots,m,m]$ which would insert a new vertex behind the last element, i.e. between $f^{-1} \cdot v$ and v.

The next stage in *generatePolygon* removes those index lists which would create a duplicate edge: *containsDuplicateEdge g f v is* checks that there are no two adjacent indices $i < j$ in *is* such that $(f^i \cdot v,\ f^j \cdot v)$ or $(f^j \cdot v,\ f^i \cdot v)$ is an edge in g (unless it is an edge in f, in which case no duplicate edge is created because f is removed). Finally the index list is turned into a list of vertices as sketched above employing

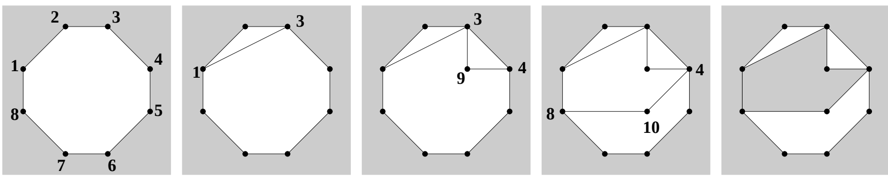
$$\textbf{datatype}\ 'a\ option = None \mid Some\ 'a$$

to distinguish an existing vertex $Some(f^i \cdot v)$ from a new vertex *None*:

$indexToVertexList\ f\ v\ is \equiv hideDups\ [f^k \cdot v.\ k \in is]$
$hideDups\ (i \cdot is) = Some\ i \cdot hideDupsRec\ i\ is$
$hideDupsRec\ a\ [] = []$
$hideDupsRec\ a\ (b \cdot bs) =$
$(\textsf{if}\ a = b\ \textsf{then}\ None \cdot hideDupsRec\ b\ bs\ \textsf{else}\ Some\ b \cdot hideDupsRec\ b\ bs)$

The result (in *generatePolygon*) is *vertexLists* of type *vertex option list list* where each list in *vertexLists* describes one possibility of inserting a final face into f.

Subdivision The last step in *generatePolygon* is to generate a new graph *subdivFace g f vos* for each *vos* in *vertexLists* by subdividing f as specified by *vos*. This is best visualized by an example. Given a face $f = [1,\ldots,8]$ and $vos = [Some\ 1,\ Some\ 3,\ None,\ Some\ 4,\ None,\ Some\ 8]$ the result of inserting a face specified by *vos* into f is shown below.



Subdividing is an iterative process where in each step we split the (remaining) face in two nonfinal faces; at the end we finalize the face. In the example we first split the face along the path $[1,3]$, then along $[3,9,4]$ and finally along $[4,10,8]$. Each splitting in two is performed by *splitFace g u v f newvs* which returns a new graph where f has been replaced by its two halves by inserting a list of new vertices *newvs* between the existing vertices u and v in f. The straightforward definition of *splitFace* is omitted.

Repeated splitting is performed by *subdivFace' g f u n vos* where u is the vertex where splitting starts, n records how many new vertices must be inserted along the seam, and *vos* is a *vertex option list* from *vertexLists*:

$subdivFace'\ g\ f\ u\ n\ [] = makeFaceFinal\ f\ g$
$subdivFace'\ g\ f\ u\ n\ (vo \cdot vos) =$
$(\textsf{case}\ vo\ \textsf{of}\ None \Rightarrow subdivFace'\ g\ f\ u\ (Suc\ n)\ vos$
$\ |\ Some\ v \Rightarrow$
$\quad \textsf{if}\ f \cdot u = v \wedge n = 0\ \textsf{then}\ subdivFace'\ g\ f\ v\ 0\ vos$
$\quad \textsf{else let}\ ws = [countVertices\ g..<countVertices\ g + n];$
$\qquad\qquad (f_1, f_2, g') = splitFace\ g\ u\ v\ f\ ws$
$\qquad \textsf{in}\ subdivFace'\ g'\ f_2\ v\ 0\ vos)$

The definition is by recursion on *vos*. The base case simply turns f into a final face in *g*. If *vos* starts with *None*, no splitting takes place but n is incremented. If *vos* starts with *Some v* there are two possibilities. Either we have merely advanced one vertex along *f*, in which case we keep on going. Or we have skipped at least one vertex of *f*, in which case we must split f between u and v, inserting n new vertices: the term $[i..<j]$ is the list of natural numbers from and including i up to but excluding *j*. Function *splitFace* returns the two new faces along with the new graph. Only f_2 is used (it is the face that is subdivided further), but returning both faces helps to state many lemmas more succinctly.

Function *subdivFace* (called from *generatePolygon*) simply starts *subdivFace′*:

$subdivFace\ g\ f\ (Some\ u \cdot vs) \equiv subdivFace'\ g\ f\ u\ 0\ vs$

Note that because all index lists produced by *enumerator* are nonempty, all vertex lists produced by *indexToVertexList* are nonempty and start with *Some*.

2.2 Invariants

Almost half the proof is concerned with verifying that *PlaneGraphs* satisfy certain invariants which are combined into the predicate *inv* :: *graph* $\Rightarrow$ *bool*. Probably half that effort is caused by showing that the extended graph representation, primarily *facesAt*, is kept consistent. The remaining properties are: each face is of size ≥ 3 and all its vertices are distinct, there are at least two faces, the faces are distinct modulo $\cong$ and if the graph has more than 2 faces also modulo reversal, the edges of distinct faces are disjoint (where edges are pairs of adjacent vertices in an oriented face), and any nonfinal face is surrounded by final faces.

3 Tame Graphs

Tameness is rooted in geometric considerations but for this paper it is simply a fixed interface to the rest of Hales' proof and should be taken as God given.

3.1 Definition of Tame Graphs

The definition relies on 4 tables a :: $nat \Rightarrow nat$, b :: $nat \Rightarrow nat \Rightarrow nat$, c :: $nat \Rightarrow int$, d :: $nat \Rightarrow nat$. Their precise definition is immaterial for this paper and can be found elsewhere [1, 5]. Like Hales (in his Java program) we have scaled all rational numbers (from the paper proof) by 1000, thus turning them into integers.

Summing over the elements of a list (below: of faces) is written $\sum x \in xs\ f\ x$. Function *faces* returns the list of faces in a graph: in our simplified model of graphs it is the identity but in the real model it is a projection.

Functions *tri* and *quad* count the number of final triangles and quadrilaterals incident to a vertex. Hales calls a face f *exceptional* if it is a pentagon or larger, i.e. if $5 \leq |vertices\ f|$. Function *except* returns the number of final exceptional faces incident to a vertex. A vertex has *type* (p, q) if $p = tri\ g\ v$, $q = quad\ g\ v$ and $except\ g\ v = 0$.

A graph is *tame* if it is plane and satisfies 8 conditions:

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

   $tame_1\ g \equiv \forall f{\in}\mathcal{F}\ g.\ 3 \leq |vertices\ f| \wedge |vertices\ f| \leq 8$

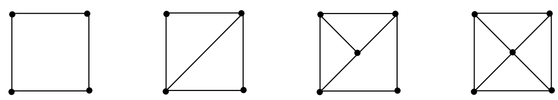
2. Every 3-cycle is a face or the opposite of a face:

   $tame_2\ g \equiv$
   $\forall\, a\ b\ c.$
   $\quad is\text{-}cycle\ g\ [a,\ b,\ c] \wedge distinct\ [a,\ b,\ c] \longrightarrow$
   $\quad (\exists f{\in}\mathcal{F}\ g.\ vertices\ f \cong [a,\ b,\ c] \vee vertices\ f \cong [c,\ b,\ a])$

   where $is\text{-}cycle\ g\ vs \equiv hd\ vs \in set\ (neighbors\ g\ (last\ vs)) \wedge is\text{-}path\ g\ vs$, function *neighbors* does the obvious and

   $is\text{-}path\ g\ [] = True$
   $is\text{-}path\ g\ (u \cdot vs) =$
   $(\mathsf{case}\ vs\ \mathsf{of}\ [] \Rightarrow True \mid v \cdot ws \Rightarrow v \in set\ (neighbors\ g\ u) \wedge is\text{-}path\ g\ vs)$

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

   The tame configurations are straightforward to describe:

   $tameConf_1\ a\ b\ c\ d \equiv [[a,\ b,\ c,\ d]]$
   $tameConf_2\ a\ b\ c\ d \equiv [[a,\ b,\ c],\ [a,\ c,\ d]]$
   $tameConf_3\ a\ b\ c\ d\ e \equiv [[a,\ b,\ e],\ [b,\ c,\ e],\ [a,\ e,\ c,\ d]]$
   $tameConf_4\ a\ b\ c\ d\ e \equiv [[a,\ b,\ e],\ [b,\ c,\ e],\ [c,\ d,\ e],\ [d,\ a,\ e]]$

   Predicate *tame-quad* formalizes that a,b,c,d forms one of the tame configurations, taking rotation into account:

   $tame\text{-}quad\ g\ a\ b\ c\ d \equiv$
   $set\ (tameConf_1\ a\ b\ c\ d) \subseteq_{\cong} vertices\ `\ \mathcal{F}\ g \vee$
   $set\ (tameConf_2\ a\ b\ c\ d) \subseteq_{\cong} vertices\ `\ \mathcal{F}\ g \vee$
   $set\ (tameConf_2\ b\ c\ d\ a) \subseteq_{\cong} vertices\ `\ \mathcal{F}\ g \vee$
   $(\exists\, e{\in}\mathcal{V}\ g - \{a,\ b,\ c,\ d\}.$
   $\quad set\ (tameConf_3\ a\ b\ c\ d\ e) \subseteq_{\cong} vertices\ `\ \mathcal{F}\ g \vee$

$set\ (tameConf_3\ b\ c\ d\ a\ e) \subseteq_{\simeq} vertices\ `\ \mathcal{F}\ g\ \vee$
$set\ (tameConf_3\ c\ d\ a\ b\ e) \subseteq_{\simeq} vertices\ `\ \mathcal{F}\ g\ \vee$
$set\ (tameConf_3\ d\ a\ b\ c\ e) \subseteq_{\simeq} vertices\ `\ \mathcal{F}\ g\ \vee$
$set\ (tameConf_4\ a\ b\ c\ d\ e) \subseteq_{\simeq} vertices\ `\ \mathcal{F}\ g)$

Finally, $tame_3$ also takes reversal of orientation into account:

$tame_3\ g \equiv$
$\forall\, a\ b\ c\ d.$
$is\text{-}cycle\ g\ [a,\ b,\ c,\ d] \wedge distinct\ [a,\ b,\ c,\ d] \longrightarrow$
$tame\text{-}quad\ g\ a\ b\ c\ d \vee tame\text{-}quad\ g\ d\ c\ b\ a$

4. The degree of every vertex is at most 6 and at most 5 if the vertex is contained in an exceptional face:

   $tame_{45}\ g \equiv \forall\, v{\in}\mathcal{V}\ g.\ degree\ g\ v \leq (\textsf{if}\ except\ g\ v = 0\ \textsf{then}\ 6\ \textsf{else}\ 5)$

   We have combined conditions 4 and 5 from [5] into one.
5. The following inequality holds:

   $tame_6\ g \equiv 8000 \leq \sum_{f \in faces\ g} \mathrm{c}\ |vertices\ f|$

6. There exists an admissible assignment of weights to faces of total weight less than 14800:

   $tame_7\ g \equiv \exists\, w.\ admissible\ w\ g \wedge \sum_{f \in faces\ g} w\ f < 14800$

   Admissibility is quite involved and discussed below. Although this is not immediately obvious, $tame_7$ guarantees there are only finitely many tame graphs. It also is the source of most complications in the proofs because it is not straightforward to check this condition.
7. There are no two adjacent vertices of type $(4, 0)$:

   $tame_8\ g \equiv \neg\ (\exists\, v{\in}\mathcal{V}\ g.\ type40\ g\ v \wedge (\exists\, w{\in}set\ (neighbors\ g\ v).\ type40\ g\ w))$

Now *tame* is the conjunction of $tame_1$ up to $tame_8$; the numbering follows [5].

Note that $tame_8$ is missing in earlier versions of the proof, e.g. www.math.pitt.edu/~thales/kepler04/fullkepler.pdf of 13/3/2004. The second author noticed and informed Hales of this discrepancy between the proof and his Java code, where the test is present. As a result Hales added $tame_8$ in the published versions of his proof.

3.2 Admissible Weight Assignment

For $w :: face \Rightarrow nat$ to be an *admissible weight assignment* it needs to meet the following requirements:

1. $admissible_1\ w\ g \equiv \forall\, f{\in}\mathcal{F}\ g.\ \mathrm{d}\ |vertices\ f| \leq w\ f$
2. $admissible_2\ w\ g \equiv$
   $\forall\, v{\in}\mathcal{V}\ g.\ except\ g\ v = 0 \longrightarrow \mathrm{b}\ (tri\ g\ v)\ (quad\ g\ v) \leq \sum_{f \in facesAt\ g\ v} w\ f$

3. $admissible_3\ w\ g \equiv$
$$\forall V.\ separated\ g\ (set\ V) \wedge set\ V \subseteq \mathcal{V}\ g \longrightarrow$$
$$\sum_{v \in V} \mathrm{a}\ (tri\ g\ v) +$$
$$\sum_{f \in [f \in faces\ g\ .\ \exists v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} \mathrm{d}\ |vertices\ f|$$
$$\leq \sum_{f \in [f \in faces\ g\ .\ \exists v \in set\ V.\ f \in set\ (facesAt\ g\ v)]} w\ f$$

The first two constraints express that d and b yield lower bounds for w. The last requirement yields another lower bound for w in terms of "separated" sets of vertices. Separatedness means that they are not neighbors, do not lie on a common quadrilateral, and fulfill some additional constraints:

$separated_1\ g\ V \equiv \forall v \in V.\ except\ g\ v \neq 0$
$separated_2\ g\ V \equiv \forall v \in V.\ \forall f \in set\ (facesAt\ g\ v).\ f \cdot v \notin V$
$separated_3\ g\ V \equiv$
$\forall v \in V.\ \forall f \in set\ (facesAt\ g\ v).\ |vertices\ f| \leq 4 \longrightarrow \mathcal{V}\ f \cap V = \{v\}$
$separated_4\ g\ V \equiv \forall v \in V.\ degree\ g\ v = 5$

Note that Hales [5] lists 4 admissibility conditions, the third of which our work shows to be superfluous.

3.3 Enumeration of Tame Graphs

The enumeration of tame graphs is a modified enumeration of plane graphs where we remove final graphs that are definitely not tame, and cut the search tree at nonfinal graphs that cannot lead to tame graphs anymore. Note that in contrast to the enumeration of plane graphs, a specification we must trust, the enumeration of tame graphs is accompanied by a correctness theorem (Theorem 3 below), the central result of the work, stating that all tame graphs are generated. Hence it is less vital to present the tame enumeration in complete detail (except to satisfy the curiosity of the reader and allow reproduceability).

The core of the tame enumeration is a filtered version of *generatePolygon*:

$generatePolygonTame\ n\ v\ f\ g = [g' \in generatePolygon\ n\ v\ f\ g\ .\ \neg\ notame\ g']$

In reality this is not the definition but the characteristic lemma. The actual definition replaces the repeated *enumeration* of lists of index lists in *generatePolygon* by a table lookup. This is "merely" an optimization for speed, but an important one. The filter *notame* removes all graphs that cannot lead to a tame graph:

$notame\ g \equiv \neg\ (tame_{45}\ g \wedge is\text{-}tame_7\ g)$
$is\text{-}tame_7\ g \equiv squanderLowerBound\ g < 14800$

Using $tame_{45}$ on nonfinal graphs is justified because the degree of a vertex can only increase as a graph is refined and because *except* takes only the final faces into account.[1] Function *squanderLowerBound* computes a lower bound for the total admissible weight of any final graph that can be generated from g. By $tame_7$ this lower bound must be < 14800.

[1] $tame_6$ on the other hand cannot be used to filter out nonfinal graphs because function c may return both positive and negative values, i.e. summing over it is not monotone under the addition of new faces.

$squanderLowerBound\ g \equiv \sum_{f \in finals\ g} \mathrm{d}\ |vertices\ f| + ExcessNotAt\ g\ None$

The lower bound consists of a d-sum over all final faces (justified by $admissible_1$ and the fact that d cannot be negative) and an error correction term *ExcessNotAt*. The definition of *ExcessNotAt* is somewhat involved and not shown. Essentially we enumerate all maximal separated sets of vertices, compute the "excess" over and above the d-sum for each one (taking a and b into account as justified by $admissible_2$ and $admissible_3$), and take the maximum. Note that the number of separated sets can grow exponentially with the number of vertices.

On top of *generatePolygonTame* we have a variant of $next\text{-}plane_p$:

$next\text{-}tame0_p\ g \equiv$
$\textsf{let}\ fs = [f{\in}faces\ g\ .\ \neg\ final\ f]$
$\textsf{in}\ \textsf{if}\ fs = []\ \textsf{then}\ []$
$\quad \textsf{else}\ \textsf{let}\ f = minimalFace\ fs;\ v = minimalVertex\ g\ f$
$\qquad \textsf{in} \bigsqcup_{i \in\ polysizes\ p\ g} generatePolygonTame\ i\ v\ f\ g$

where *polysizes* restricts the possible range $[3..p + 3]$ to those sizes which can still lead to tame graphs:

$polysizes\ p\ g \equiv [n{\in}[3..p + 3]\ .\ squanderLowerBound\ g + \mathrm{d}\ n < 14800]$

The justification is that the insertion of a new n-gon into g adds at least d n to *squanderLowerBound g*. Hence all these graphs would immediately be discarded again by *notame* and *polysizes* is merely an optimization, but one which happens to reduce the run time by a factor of 10.

The key correctness theorems for *squanderLowerBound* (recall *inv* from §2.2) are that it increases with *next-tame0* (in fact with *next-plane*)

Theorem 1. If $g' \in set\ (next\text{-}tame0_p\ g)$ and *inv g* then $squanderLowerBound\ g \leq squanderLowerBound\ g'$.

and for (final) tame graphs *squanderLowerBound* is a lower bound for the total weight of an admissible assignment:

Theorem 2. If *tame g* and *final g* and *inv g* and *admissible w g* and $\sum_{f \in faces\ g} w\ f < 14800$ then $squanderLowerBound\ g \leq \sum_{f \in faces\ g} w\ f$

These two theorems are the main ingredients in the completeness proof of *next-tame0* w.r.t. *next-plane*: any tame graph reachable via *next-plane* is still reachable via *next-tame0*.

Now we compose *next-tame0* with a function *makeTrianglesFinal* (details omitted) which finalizes all nonfinal triangles introduced by *next-tame0*:

$next\text{-}tame1_p \equiv map\ makeTrianglesFinal \circ next\text{-}tame0_p$

This step appears to be a trivial consequence of $tame_2$ which says that all 3-cycles must be triangles, i.e. that one should not be allowed to subdivide a triangle further. The latter implication, however, is not completely trivial: one has to show that if one ever subdivides a triangle, that triangle cannot be re-introduced as a face later on. A lengthy proof yields completeness of *next-tame1* w.r.t. *next-tame0*. The invariants (§2.2) are absolutely essential here.

As a final step we filter out all untame final graphs:

$\textit{next-tame}_p \equiv \textit{filter}\ (\lambda g.\ \neg\ \textit{final}\ g \lor \textit{is-tame}\ g) \circ \textit{next-tame1}_p$
$\textit{is-tame}\ g \equiv \textit{tame}_{45}\ g \land \textit{tame}_6\ g \land \textit{tame}_8\ g \land \textit{is-tame}_7\ g \land \textit{is-tame}_3\ g$

Tameness conditions 1 and 2 are guaranteed by construction. Conditions 45, 6 and 8 are directly executable. Condition 7 has been discussed already. This leaves condition 3, the check of all possible 4-cycles:

$\textit{is-tame}_3\ g \equiv$
$\forall\ vs{\in}set\ (\textit{find-cycles}\ 4\ g).$
$\quad \textit{is-cycle}\ g\ vs \land \textit{distinct}\ vs \land |vs| = 4 \longrightarrow \textit{ok4}\ g\ vs$

This implementation is interesting in that it employs a search-and-check technique: function *find-cycles* need not be verified at all because the rest of the code explicitly checks that the vertex lists are actually cycles of distinct vertices of length 4. This can at most double the execution time but reduces verification time. In fact, *find-cycles* is expressed in terms of a *while*-functional [9] which simplifies definition but would complicate verification. Function *ok4* is a straightforward implementation of *tame-quad*.

Note that this search-and-check technique is applicable only because we merely need to ensure completeness of the enumeration of tame graphs, not correctness. Otherwise we would need to verify that *find-cycles* finds all cycles.

Completeness of *next-tame* w.r.t. *next-tame1* follows from Theorem 2 together with the implementation proof of *ok4* w.r.t. *tame-quad*. Putting the three individual completeness theorems together we obtain the overall completeness of *next-tame*: all tame graphs are enumerated.

Theorem 3. If $\textit{Seed}_p$ $[\textit{next-plane}_p]\rightarrow^*$ g and *final* g and *tame* g then $\textit{Seed}_p$ $[\textit{next-tame}_p]\rightarrow^*$ g.

The set of tame graphs is defined in the obvious manner:

$\textit{TameEnum}_p \equiv \{g \mid \textit{Seed}_p\ [\textit{next-tame}_p]\rightarrow^*\ g \land \textit{final}\ g\}$
$\textit{TameEnum} \equiv \bigcup_{p \le 5} \textit{TameEnum}_p$

An executable version of $\textit{TameEnum}_p$ is provided under the name *tameEnum*. It realizes a simple work list algorithm directly on top of *next-tame* and need not be shown or discussed, except for one detail. Being in a logic of total functions we have to apply an old trick: since we want to avoid proving termination of the enumeration process (which is bound to be quite involved), *tameEnum* takes two parameters: the usual p and a counter which is decremented in each step. If it reaches 0 prematurely, we return *None*, otherwise we return *Some Fs* where *Fs* is the collected list of final graphs, the result of the enumeration. When running *tameEnum* we merely need to start with a large enough counter. Because the returned graphs are all final we reduce each graph to a list of list of vertices via

$\textit{fgraph}\ g \equiv \textit{map vertices}\ (\textit{faces}\ g)$

before including it in the result. Hence the actual return type of *tameEnum* is *vertex fgraph list* where

$$'a\ \textit{fgraph} = {}'a\ \textit{list list}.$$

We merely show *tameEnum*'s correctness theorem, not the definition:

Theorem 4. If $tameEnum\ p\ n = Some\ Fs$ then $set\ Fs = fgraph$ ' $TameEnum_p$.

As a final step we need to run *tameEnum p n* with suitably large n (such that *Some* is returned) for all $p \leq 5$ [2] and compare the result with the contents of the Archive.

4 The Archives

It turned out that Hales' Archive was complete but redundant. That is, our verified enumeration produced only 2771 graphs as opposed to Hales' 5128. The reason is twofold: there are many isomorphic copies of graphs in his Archive and it contains a number of non-tame graphs (partly because for efficiency reasons he does not enforce $tame_3$ completely in his Java program). The new reduced Archive can be found at the first author's web page.

The new Archive is a constant *Archive* :: *vertex fgraph set* in the Isabelle theories which is defined via the concatenation of 6 separate archives, one for each $p \leq 5$:

$Archive \equiv set\ (Tri\ @\ Quad\ @\ Pent\ @\ Hex\ @\ Hept\ @\ Oct)$

The main theorem of our work is the completeness of *Archive*:

Theorem 5. If $g \in PlaneGraphs$ and $tame\ g$ then $fgraph\ g \in_{\simeq} Archive$.

Relation $\simeq$ is graph isomorphism (§4.1) and $x \in_{\simeq} M \equiv \exists y{\in}M.\ x \simeq y$. This theorem is a combination of the completeness of *next-tame* (Theorem 3) and of *fgraph* ' $TameEnum \subseteq_{\simeq} Archive$ (where $M \subseteq_{\simeq} N \equiv \forall x{\in}M.\ x \in_{\simeq} N$). The latter is a corollary of the fact that, for each $p \leq 5$, *tameEnum p n* (for suitable n) returns *Some Fs* such that *Fs* is equivalent to the corresponding part of the Archive. Quite concretely, we have proved by evaluation (§1.3) that *same* (*tameEnum 0 800000*) *Tri*, *same* (*tameEnum 1 8000000*) *Quad*, *same* (*tameEnum 2 20000000*) *Pent*, *same* (*tameEnum 3 4000000*) *Hex*, *same* (*tameEnum 4 1000000*) *Hept*, and *same* (*tameEnum 5 2000000*) *Oct*, where *same* is an executable check of equivalence (modulo $\simeq$)[3] of two lists of *fgraphs*. Corollary *fgraph* ' $TameEnum \subseteq_{\simeq} Archive$ follows by correctness of *tameEnum* (Theorem 4) .

We cannot detail the definition of *same* (or its correctness theorem) but we should point out that it is a potential bottleneck: for $p = 2$ we need to check 15000 graphs for inclusion in an archive of 1500 graphs — modulo graph isomorphism! Although isomorphism of plane graphs can be determined in linear time [8], this algorithm is not very practical because of a large constant factor. Instead we employ a hashing scheme to home in on the isomorphic graph quickly.

[2] By $tame_1$ we are only interested in graphs where all faces are of size $\leq 8 = 5{+}3$, the 3 being added in *next-plane*.

[3] A check for $\subseteq_{\simeq}$ would suffice but it is nice to know that *Archive* is free of junk.

The graphs of each archive are stored in a search tree (a *trie*) indexed by a list of natural numbers. The index is the concatenation of a number of hash values invariant under isomorphism. The most important component is obtained by adding up, for each vertex, the size of the faces around that vertex, and then sorting the resulting list. This idea is due to Hales.

4.1 Plane Graph Isomorphisms

For lack of space we present our definition of isomorphism but not its implementation:

$$\begin{aligned}
&\textit{is-pr-iso}\ \varphi\ \textit{Fs}_1\ \textit{Fs}_2 \equiv \textit{is-pr-Iso}\ \varphi\ (\textit{set Fs}_1)\ (\textit{set Fs}_2)\\
&\textit{is-pr-Iso}\ \varphi\ \textit{Fs}_1\ \textit{Fs}_2 \equiv \textit{is-Hom}\ \varphi\ \textit{Fs}_1\ \textit{Fs}_2 \wedge \textit{inj-on}\ \varphi\ (\bigcup_{F \in \textit{Fs}_1}\ \textit{set}\ F)\\
&\textit{is-Hom}\ \varphi\ \textit{Fs}_1\ \textit{Fs}_2 \equiv \textit{map}\ \varphi\ \text{`}\ \textit{Fs}_1\ //\ \{\cong\} = \textit{Fs}_2\ //\ \{\cong\}
\end{aligned}$$

Parameter φ is a function of type *vertex* $\Rightarrow$ *vertex*. Predicate *is-pr-iso* compares lists of faces (*fgraphs*), *is-pr-Iso* sets of faces; *pr* stands for *proper*. Proper isomorphisms assume that the faces of both graphs have the same orientation. An isomorphism is defined as usual as an injective homomorphism. A homomorphism must turn one graph into the other, modulo rotation of faces: the infix // is quotienting and the symbol $\{\cong\}$ is defined as $\{(f_1, f_2) \mid f_1 \cong f_2\}$.

'Improper' isomorphisms allow to reverse the orientation of all faces in one graph (*rev* reverses a list):

$$\begin{aligned}
&\textit{is-iso}\ \varphi\ \textit{Fs}_1\ \textit{Fs}_2 \equiv \textit{is-Iso}\ \varphi\ (\textit{set Fs}_1)\ (\textit{set Fs}_2)\\
&\textit{is-Iso}\ \varphi\ \textit{Fs}_1\ \textit{Fs}_2 \equiv \textit{is-pr-Iso}\ \varphi\ \textit{Fs}_1\ \textit{Fs}_2 \vee \textit{is-pr-Iso}\ \varphi\ \textit{Fs}_1\ (\textit{rev}\ \text{`}\ \textit{Fs}_2)
\end{aligned}$$

Two *fgraphs* are isomorphic if there exists an isomorphism between them:

$$g_1 \simeq g_2 \equiv \exists \varphi.\ \textit{is-iso}\ \varphi\ g_1\ g_2$$

5 Statistics

The starting point were 2200 lines of Java, the result 600 lines of executable HOL (= ML), excluding comments, debugging aids, and libraries. This reduction is primarily due to simplifications of the algorithm itself: not splitting the treatment of triangle and quadrilateral seed graphs into 17 cases, dropping all symmetry checks, dropping the special treatment of nonfinal quadrilaterals, and dropping some complicated lower bound estimates (which are all still present in [1]). The simplicity of the final solution belies the difficulty of arriving at it.

The whole formalization encompasses 17000 lines of definitions and proofs. Running the complete proof takes 165 minutes on a Xeon: the completeness proof takes 15 minutes, evaluating the enumeration 105 minutes, and comparing the resulting graphs with the Archive (modulo graph isomorphism) 45 minutes.

During execution of the enumeration, the gargantuan number of 23 million graphs are generated and examined, of which 35000 are final. The distribution of graphs over the new Archive (for $p = 0, \ldots, 5$) is (20,22,13), (923,18,12), (1545,18,13), (238,17,12), (23,16,12), and (22,15,12), where each triple gives the number of graphs, average number of faces, and average number of vertices for that group of graphs.

6 Future Work

The enumeration of plane graphs needs to be connected with some abstract notion of planarity. Hales is preparing a revised proof based on hypermaps that could serve as the glue — face lists are easily turned into hypermaps. On the other end, it needs to be shown that none of the tame graphs constitutes a counter example. The linear programming techniques for this step are in place [10], but their application is nontrivial and not well documented in Hales proof.

Finally there is the exciting prospect of modifying our proof to cover a very similar graph enumeration in the proof of the *Dodecahedral Conjecture* [7].

Acknowledgments The first author wishes to thank: Tom Hales for generously hosting his sabbatical semester at the University of Pittsburgh and for patiently answering all questions; Jeremy and Sean for lunches and friendship; Q & U for late night entertainment.

References

1. G. Bauer. *Formalizing Plane Graph Theory — Towards a Formalized Proof of the Kepler Conjecture*. PhD thesis, Technische Universität München, 2006.
2. 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.
3. G. Gonthier. A computer-checked proof of the four colour theorem. Available at research.microsoft.com/~gonthier/4colproof.pdf.
4. T. C. Hales. Cannonballs and honeycombs. *Notices Amer. Math. Soc.*, 47:440–449, 2000.
5. T. C. Hales. A proof of the Kepler conjecture. *Annals of Mathematics*, 162:1063–1183, 2005.
6. T. C. Hales. Sphere packings, VI. Tame graphs and linear programs. *Discrete and Computational Geometry*, 2006. To appear.
7. T. C. Hales and S. McLaughlin. A proof of the dodecahedral conjecture. E-print archive arXiv.org/abs/math.MG/9811079.
8. J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *STOC '74: Proc. 6th ACM Symposium Theory of Computing*, pages 172–184. ACM Press, 1974.
9. 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/.
10. S. Obua. Proving bounds for real linear programs in Isabelle/HOL. In J. Hurd, editor, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lect. Notes in Comp. Sci.*, pages 227–244. Springer-Verlag, 2005.
11. R. Zumkeller. A formalization of global optimization with Taylor models. In *Automated Reasoning (IJCAR 2006)*, 2006. This volume.