A computer-checked proof of the Four Color Theorem

Georges Gonthier

A computer-checked proof of the Four Color Theorem

Georges Gonthier
Microsoft Research Cambridge

This report gives an account of a successful formalization of the proof of the Four Color Theorem, which was fully checked by the Coq v7.3.1 proof assistant [13]. This proof is largely based on the mixed mathematics/computer proof [26] of Robertson *et al.*, but contains original contributions as well. This document is organized as follows: section 1 gives a historical introduction to the problem and positions our work in this setting; section 2 defines more precisely what was proved; section 3 explains the broad outline of the proof; section 4 explains how we exploited the features of the Coq assistant to conduct the proof, and gives a brief description of the tactic shell that we used to write our proof scripts; section 5 is a detailed account of the formal proof (for even more details the actual scripts can be consulted); section 6 is a chronological account of how the formal proof was developed; finally, we draw some general conclusions in section 7.

1 The story

The Four Color Theorem is famous for being the first long-standing mathematical problem to be resolved using a computer program. The theorem was first conjectured in 1852 by Francis Guthrie, and after over a century of work by many famous mathematicians [36,28] (including De Morgan, Peirce, Hamilton, Cayley, Birkhoff, and Lebesgue), and many incorrect "proofs", it was finally proved by Appel and Haken in 1976 [3]. This proof broke new ground because it involved using computer programs to carry out a gigantic case analysis, which could not, in any real sense, be performed by hand: it covered, quite literally, a billion cases.

The Appel and Haken proof attracted a fair amount of criticism. Part of it concerned the proof style: the statement of the Four Color Theorem is simple and elegant so many mathematicians expected a simple and elegant proof that would explain, at least informally, *why* the theorem was true — not opaque IBM 370 assembly language programs [5]. Another part, however, was more rational skepticism: computer programming is known to be error-prone, and difficult to relate precisely to the formal statement of a mathematical theorem. The fact that the proof also involved an initial manual case analysis [4] that was large (10,000 cases), difficult to verify, and in which several small errors were detected, also contributed to the uncertainty.

In 1995, Robertson, Sanders, Seymour, and Thomas published a more streamlined proof of the theorem [26], using an argument quite similar to the one used by Appel and Haken, but employing C programs to check both the giant and the large case analyses. This second proof, along with the edition of a definitive monograph for the original proof [6], cleared up most doubts concerning the truth of the theorem. Furthermore, Robertson *et al.* used the larger computational resources available to them to search for and find a significantly (four times) smaller set of cases for the second analysis, by systematically exploring variants in the initial case analysis.

Our work can be seen as an ultimate step in this clarification effort, completely removing the two weakest links of the proof: the manual verification of combinatorial arguments, and the manual verification that custom computer programs correctly fill in parts of those arguments. To achieve this, we have written a *formal proof script* that covers both the mathematical and computational parts of the proof. We have run this script through the Coq proof checking system [13,9], which mechanically verified its correctness in all respects. Hence, even though the correctness of our proof still depends on the correct operation of several computer hardware and software components (the processor, its operating system, the Coq proof checker, and the OCaml compiler that compiled it), none of these components are specific to the proof of the Four Color Theorem. All of them come off-the-shelf, fulfill a more general purpose, and can be (and are) tested extensively on numerous other jobs, probably much more than the mind of an individual mathematician reviewing a proof manuscript could ever be. In addition, the most specific component we use — the Coq system, to which the script is tuned — can output a *proof witness*, i.e., a longhand detailed description of the chain of formal logical steps that was used in the proof. This witness can, in principle, be checked independently (technically, it is a term in a higher-order lambda calculus). Because this witness records only logical steps, and not computation steps, its size remains reasonable, despite the large amount of computation needed for actually checking the proof.

Although this work is purportedly about using computer programming to help doing mathematics, we expect that most of its fallout will be in the reverse direction — using mathematics to help programming computers. The approach that proved successful for this proof was to turn almost every mathematical concept into a data structure or a program, thereby converting the entire enterprise into one of program verification. Most of our work consisted in experimenting with and developing proof techniques for dealing with such "higher-level" programs, leading to a proof style that departs significantly from common practice in machine-assisted theorem proving. For instance, we have almost no use for decision procedures or complex proof tactics, and insert relatively few explicit intermediate statements. In fact, many of our proofs look more like debugger or testing scripts than mathematical arguments. We believe that the fact that the "highbrow" mathematics involved in the Four Color Theorem can be dealt with effectively in this fashion indicates that there is considerable scope for using more effectively mathematical abstraction in software engineering.

2 The statement of the theorem

Part of the attractiveness of the Four Color Theorem is its nutshell-sized statement:

> **Four Color Theorem**: *Any planar map can be colored with only four colors*.

Of course, this is somewhat of a cheat, and we should be more explicit about what we mean by "colored map":

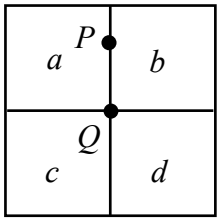
> **Four Color Theorem**: *The regions of any simple planar map can be colored with only four colors, in such a way that any two adjacent regions have different colors.*

This still leaves the terms *simple planar map*, *region*, and *adjacent* open; in formal mathematics, even such intuitive graphical notions must be precisely defined. It is possible to give a quite general definition, using some basic notions in abstract topology:

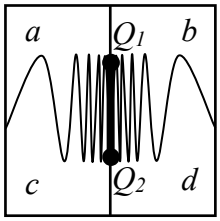
> **Definition**: *A* planar map *is a set of pairwise disjoint subsets of the plane, called* regions*. A* simple *map is one whose regions are connected open sets.*
> **Definition:** *Two regions of a map are* adjacent *if their respective closures have a common point that is not a corner of the map.*
> **Definition:** *A point is a corner of a map if and only if it belongs to the closures of at least three regions.*

Without the restriction to simple maps in the first definition we would have the Empire Coloring Problem, where regions can consist of two or more separate pieces; such disjointed maps can require more than four colors (up to $6r+1$, to be exact, when regions are allowed to have up to r disjoint open parts each).

The second definition is just a formal way of saying that regions are adjacent when they "touch" at points other than corners. For example, consider a map consisting of four regions *a*, *b*, *c*, *d* that are the interior of the four squares below



Then the closure of *a* is the union of *a* with its framing square, similarly for *b*, *c*, and *d*. The closures of *a* and *b* have a common point *P* that is not in the closure of *c* or *d*, so *a* and *b* are adjacent. On the other hand, *a* and *d* are *not* adjacent, since the only common point of their respective closures is *Q*, and *Q* also belongs to the closures of *b* and *d*. Note again that without this restriction more than four colors could be needed, e.g., if *a*, *b*, *c*, and *d* were all mutually adjacent, the map obtained by adding the outside of the large square would require five colors. Note also that "corners" are not necessarily isolated points: replace the horizontal line segments in the map above with two arcs of the curve $y = \sin {}^{1}/x$, and there's an entire line segment $[Q_1Q_2]$ of "corners" all touching each of *a*, *b*, *c*, and *d*.



The other notions we used in the above – open sets, connected sets, closures – are all standard basic undergraduate material. Fully formalizing these notions along with the definitions above takes up about 30 lines of our formal proof, in the file `realmap.v`. But, precisely because this is a *formal* proof, these few lines, along with the formal definition of the real numbers in file `real.v` (about 200 lines in total) are all that one needs to read and understand in order to ascertain that the statement below in file `fourcolor.v` is indeed a proof of the Four Color Theorem:

```
Variable R : real_model.
Theorem four_color (m : map R) :
  simple_map m -> map_colorable 4 m.
Proof.
exact: compactness_extension four_color_finite.
Qed.
```

The other 60,000 or so lines of the proof can be read for insight or even entertainment, but need not be reviewed for correctness. That is the job of the Coq proof assistant, a job for computers.

The definition we use for the real numbers is both axiomatic and constructive, following the approach used in the C-Corn library for intuitionistic real numbers [17]. We define a `real_model` as a type of objects with operations that satisfy a fairly standard set of axioms[1] for the real numbers and prove the Four Color Theorem for an arbitrary `real_model`. However, we also provide an explicit construction of a model (file `dedekind.v`) and a proof that this model is unique up to isomorphism (file `realcategorical.v`). Thus, it is not necessary to read through the somewhat involved Dedekind cut construction of the real model to ascertain that the `R` in the statement of `four_color` indeed stands for the usual real numbers. Nevertheless, the existence of the construction ensures that the real axioms we use are consistent with the Coq type theory[2] [13,2,35,8,24].

Although the statement of the Four Color Theorem uses notions from Analysis, the Four Color Theorem is essentially a result in Combinatorics: the essence of the theorem is a statement of properties of certain finite arrangements of finite objects. Indeed, most mathematical papers on the subject pay only lip service to the continuous statement and quickly (and informally) rephrase the problem in graph theory: coloring a map is trivially equivalent to coloring the graph obtained by taking the regions of the map as nodes, and linking every pair of adjacent regions.

In principle, it is even possible to completely forget that regions are sets of points in the graph problem, but this is never done in practice, because the purely graph theoretic characterizations of graphs generated by planar maps, such as the Kuratowski minor exclusion [23] or the Hopcroft-Tarjan planarity test [20], are not intuitively compelling and are difficult to use in proofs. Instead, the planarity of a graph is characterized by means of a planar embedding that maps nodes to points and edges to pairwise disjoint continuous arcs (the justification for this mapping is often informal). It is then possible to apply the

[1] There are a few twists due to the fact that Coq is based on intuitionistic type theory rather than classical set theory. The main consequence is that we have to state all the arithmetic axioms, such as the commutativity of +, using the extensional equality of the structure (derived from the inequality preorder by $x = y \triangleq x \leq y \wedge x \geq y$), rather than Coq's primitive intensional equality; we would need additional extensionality axioms to construct the corresponding quotient type in Coq. Also, we had to pick a form of the supremum axioms that implies the excluded middle axiom, $\forall P.\ P \vee \neg P$

[2] The axiomatization of the reals in the Coq standard library includes a ceiling function that maps the reals to the inductive integers, and whose consistency with the impredicative type theory of Coq v7 is questionable. Our construction relies only on an excluded middle assumption, whose consistency has been well studied.

> **Jordan Curve Theorem**: *The complement of any Jordan (i.e., closed continuous) curve is the disjoint union of two components, each of whose frontier is the whole curve.*

to make use of the planarity assumption, as every cycle in the graph is mapped to a Jordan curve by the embedding. However, this approach results in proofs that, while convincing for humans, are difficult to formalize because they contain an informal mix of combinatorics and topology.

We therefore decided to use a different kind of combinatorial structure, known as *hypermaps*[15,31,32,16], to carry out the combinatorial core of the proof; hypermaps will be described in detail in section 5.1. This design decision had several advantages:

- We could formulate a self-contained combinatorial theorem, and prove it under weaker logical assumptions (using only intuitionistic logic).
- The use of Analysis was circumscribed to the discretization of the problem, and even this could be carried out directly by elementary methods, without appealing to the Jordan Curve Theorem (in fact, the discretization does not follow from that Theorem, so it is somewhat off topic).

The second point is important because the Jordan Curve Theorem is a rather difficult result[3], whose known proofs rely heavily on either complex analysis and/or homology. The approach we have outlined above allows us to avoid the Jordan Curve Theorem, although we do use a combinatorial analogue (see Section 5.1).

3 A historical overview of the proof

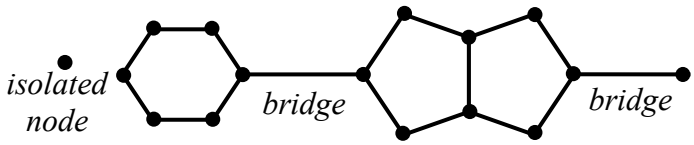
Aside from the hypermap formulation and the preliminary discretization lemma, our proof follows very closely the 1994 proof by Robertson, Sanders, Seymour and Thomas [26,27]; in turn, that proof follows the same outline as the 1976 proof by Appel and Haken [4,5], whose general principles go back to the original (incorrect) 1879 proof by Kempe[22], with important contributions by Birkhoff[10], Heesch[18], and many others (see [36] for a full history). In presenting the general idea of the proof, we can only pay tribute to the ingenuity of our predecessors; our actual work was to find a systematic way of getting the details right – *all* of them.

Our (combinatorial) hypermap coloring theorem can be interpreted as a special case of the general Four Color Theorem map coloring theorem, so we shall continue to use the map coloring formulation here, even though most of the works we reference use the graph formulation introduced in Section 2. We shall make the combinatorial interpretation evident by focusing on a particular class of maps:

> **Definition**: *A* polygonal outline *is the pairwise disjoint union of a finite number of open line segments, called* edges*, with a finite set of* nodes *that contains the endpoints of the edges. The regions of a* finite polygonal planar map*, called* faces*, are the connected components of the complement of a polygonal outline.*

[3] The machine formalisation of the Jordan Curve Theorem was the subject of several projects; T. Hales has just announced that he had completed the proof using HOL Light. There seemed to be little point in trying to replicate these works.

Finite polygonal maps correspond quite closely to hypermaps, and indeed the construction used to prove our discretization lemma can also be used to show that the restriction to finite polygonal maps entails no loss of generality. On the contrary, the definition of polygonal maps is slightly too broad. It allows isolated nodes, disconnected outlines, and extraneous edges that are bordered by the same face on either side, called *bridges*.



The discretization construction avoids such degeneracies; it generates polygonal maps that are *bridgeless* and *connected*. These properties play an important role in the combinatorial proof; indeed, the exact statement of our combinatorial theorem is

```
Theorem four_color_hypermap (g : hypermap) :
 planar_bridgeless g -> four_colorable g.
```

The faces of a finite bridgeless connected polygonal map are the interior of simple polygons (and are thus bounded), except one, which is the exterior of a simple polygon (and is thus unbounded). The edges and nodes of such a map are the sides and vertices of these polygons, respectively. Thus, the map is the planar projection of a polyhedron, hence the following

Definition: *A* polyhedral map *is a* finite *bridgeless connected polygonal map.*

The number N of nodes, F of faces, and E of edges of a polyhedral map are related by the well-known *Euler formula*

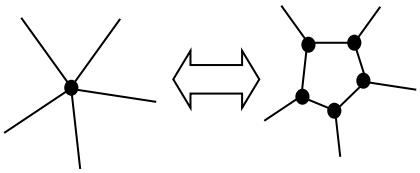
$$N + F - E = 2$$

This formula characterizes planarity. We use it to define planar hypermaps; in the discretization construction, it is established by counting points, lines and squares on a rectangular grid.

In the rest of this section, we give an informal outline of the combinatorial proof, transposed to polyhedral maps. To avoid spurious computational geometry considerations, we will allow edges to be broken (as opposed to straight) line segments; this is only apparent in the "digon" case in c) below.

The basic proof outline of the Four Color Theorem, oddly, has not changed since the initial, incorrect proof attempt [22] by Kempe in 1879. Kempe's proof went as follows:

a) It is enough to consider only *cubic* maps, where exactly three edges meet at each node: just cover each node with a new polygonal face; if the resulting cubic map can be colored, then simply deleting the faces that were added yields a coloring of the original map.

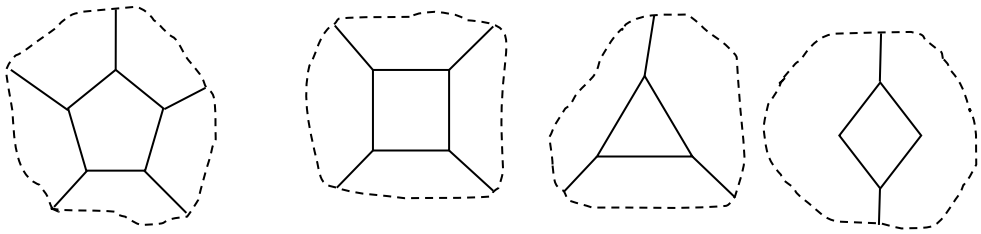


b) In a cubic map, we have $3N = 2E$ edge endpoints, so the Euler formula can be rewritten $2E = 6F - 12$. Since 2E is also the total number of sides of all faces, this states that every face has on average almost six sides, but that globally there

are 12 missing sides (this is why it takes 12 pentagons, along with the hexagons, to stitch a football together).

c) Consider a *minimal* cubic counter example to the Four Color Theorem: assuming there are polyhedral maps that require at least five colors, pick a cubic one with the *smallest* number of faces.

d) By b) this minimal counter example map must have a face with five sides or less, so the neighborhood around that face must look like one of these



If we erase one or two well-chosen sides of the central face, we get a smaller cubic polyhedral map, which must therefore be four colorable. (For the square and pentagon, we erase two sides x and y, chosen so that the faces on the other side of x and y, respectively, are neither equal nor adjacent.)

e) If the central face is not a pentagon, we can immediately find a color for it (recalling that for a square, we erase two opposite sides).

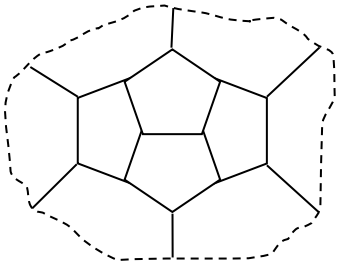
f) If the central face is a pentagon, we may need to modify the map coloring so that a color is free for the pentagon. Roughly,[4] this is done by locally interchanging two colors in any group of faces that is surrounded by the other two colors. Such two-toned groups (called "Kempe chains") cannot cross each other, since the map is planar. Kempe enumerated all the possible ways in which chains could connect the faces in the ring surrounding the central pentagon to show that recoloring was always possible.

The error that took ten years to spot and almost a century to fix occurred in the last step — some cases were missing in the enumeration (Kempe failed to note that interchanging colors in one chain may scramble other chains).

The correct proof follows the same principle, but considers larger map fragments, called *configurations*. A configuration consists of a connected group of (whole) faces, called its *kernel*, surrounded by a *ring* of partial faces. The first major steps towards a correct proof were made by Birkhoff in 1913[10]. He showed that

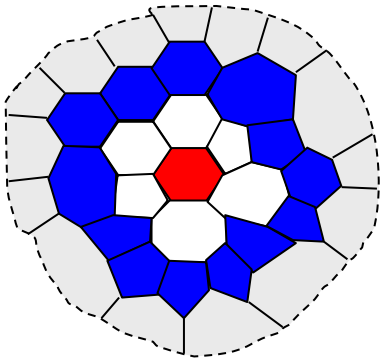
g) For some *reducible* configurations, such as the one below whose kernel is a group of four pentagons (known as the Birkhoff diamond), the Kempe argument f) that failed for single pentagons could be carried out soundly.

[4] See section 5.2 for a precise description of the version of this operation that is used in the formal proof.

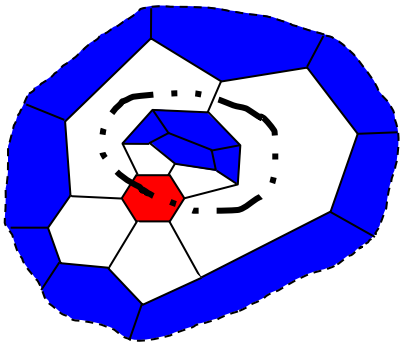


h) The Kempe argument is sound for any configuration surrounded by a ring of at most five faces—*except* the one containing a single pentagon. Let us say that a ring is *trivial* when its interior is either empty or consists of a single pentagon; then it follows from the above that a minimal counter example can contain no non-trivial rings of length 5 or less: it must be *internally 5-connected.* We refer to this result as the "Birkhoff Lemma" in the sequel.

i) As a consequence of h), in a minimal counter example, every face is surrounded by two concentric *simple* rings, i.e., its *second neighborhood* must have a shape similar to



This excludes, for instance, maps where two non-consecutive neighbors of the central face are adjacent, as in the figure below, because such maps contain a configuration with a three-face ring (indicated by the heavy mixed dash).



We refer to this result as the "Birkhoff Theorem" in the sequel.

These results suggest the following strategy for proving the Four Color Theorem:

Find an explicit list *R* of configurations such that one can establish

I. *reducibility*: every configuration in *R* is reducible, using some variant of g).
II. *unavoidability*: every minimal counter example must contain at least one of the configurations in *R*, using b), e), h), and i).

The Four Color Theorem then follows by contradiction, since no reducible configuration should occur in a minimal counter example.

Although much of the work that followed focused on finding a suitable *R*, for us it was a given: we just used the list of 633 configurations devised by Robertson *et al.* [26].

The methods used in g), with some generalizations and minor improvements, are sufficient for part I, reducibility. It was only in 1969, however, that a systematic method for part II, unavoidability, was published by Heesch[5][18]. The method, called *discharging*, can be outlined as follows:

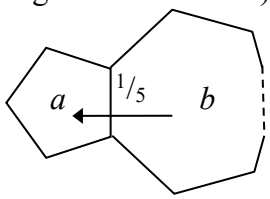
1) Find a formula for computing the "average" *arity* (number of sides) of the polygons in the second neighborhood of a face, such that the sum of these averages over all faces is equal to 2*E*, the total number of sides in the graph.
2) Show that in a minimal counter example, every second neighborhood whose average arity, computed using the formula of 1), is strictly *less* than 6, contains one of the reducible configurations in *R*.

Unavoidability follows from 2), since by b) and 1) any cubic map must contain some second neighborhood whose average arity is less than 6. By i) step 2) can be carried out by enumerating the possible arities of the faces in a second neighborhood. The average arity constraint usually provides an upper bound for this enumeration, and this bound is quite tight, since by e) all arities are at least 5. Thus, the total size of the enumeration is relatively small (in the 10,000 range); furthermore, the neighborhoods produced by the enumeration will have a high density of pentagons and hexagons, and thus stand a good chance of being reducible.

Heesch also gave a method for constructing suitable averaging formulas for step 1). To ensure that the sum-of-averages constraint in 1) is always met, the averaging process is described in terms explicit transfers of (fractions of) arity between neighboring faces: the average arity $\overline{\delta}(a)$ of the neighborhood a face *a* is computed as

$$\overline{\delta}(a) = \delta(a) + \sum_b T_{ba} - \sum_b T_{ab}$$

where $\delta(a)$ is the arity of a, and T_{ba} is the fraction of arity transferred to *a* from one of its neighbors *b*. Heesch defined the value of T_{ba} with a set of explicit patterns, called *discharge rules*. Each rule specifies that a fraction of arity should be transferred from *b* to *a* if their common neighborhood fits its pattern, and T_{ba} is the sum of all these fractions. For instance (following Robertson *et al*.) we use the following rule



This transfers to a pentagon one fifth of a side from any adjacent face that has more than five sides. Therefore, the neighborhood of any pentagon that is not adjacent to another pentagon will have an average arity of *exactly* 6. Robertson *et al*. showed [26] that with a handful of similar rules, this property can be extended to all pentagons and hexagons: unless they are part of a reducible clump of pentagons and hexagons, their average arity is exactly 6. In other terms, the "missing side" of each pentagon is *exactly* borrowed from the heptagons, octagons, and polygons of higher arity in its second neighborhood. Since each rule transfers only a fraction of a side, the second neighborhood of, say, an octagon with an average arity of less than 6 must contain many pentagons, which increases the likelihood that it will contain a reducible configuration.

[5] Heesch had been developing and using the methods since the 1940s, however.

Unlike a weighted arithmetic average, the average arity obtained by discharging might not depend on the arities of all the faces in the neighborhood, so it may not be possible to enumerate the arities of *all* the faces in a neighborhood with an average arity less than 6. This is not a problem in practice, since the enumeration can be stopped as soon as a reducible configuration occurs, and the set of discharge rules can be adjusted so that this is always the case. Discharging also gives bounds on the size of neighborhoods that need to be enumerated. The rules proposed by Robertson *et al.* never transfer more than half a side between two faces, so there is no need to enumerate the neighborhoods of dodecagons and larger polygons, since their average arity is never less than 6.

Although Heesch had correctly devised the plan of the proof of the Four Color Theorem, he was unable to actually carry it out because he missed a crucial element: computing power. The discharge rules he tried gave him a set R containing configurations with a ring of size 18, for which checking reducibility was beyond the reach of computers at the time. However, there was hope, since both the set of discharge rules and the set R could be adjusted arbitrarily in order to make every step succeed.

Appel and Haken cracked the problem in 1976 by focusing their efforts on adjusting the discharge rules rather than extending the reducible configuration set R, using a heuristic due to Heesch for predicting whether a configuration would be reducible (with 90% accuracy), without performing a full check. By trial and error, they arrived at a set R containing only configurations of ring size at most 14, for which they barely had enough computing resources to do the reducibility computation. Indeed, the discharging formula had to be complicated in many ways in order to work around computational limits. In particular the formula had to transfer arity between non-adjacent faces, and to accommodate this extension unavoidability had to be checked manually. It was only with the 1994 proof by Robertson *et al.* that the simple discharging formula that Heesch had sought was found.

We took the work of Robertson *et al.* as our starting point, reusing their optimized catalogue of 633 reducible configurations, their cleverly crafted set of 32 discharge rules, and their branch-and-bound enumeration of second neighborhoods [27]. However, we had to devise new algorithms and data structures for performing reducibility checks on configurations and for matching them in second neighborhoods, as the C integer programming coding tricks they used could not be efficiently replicated in the context of a theorem prover, which only allows pure, side effect free data structures (e.g., no arrays). And performance *was* an issue: the version of the Coq system we used needed three days to check our proof, whereas Robertson *et al.* only needed three hours… ten years ago! (Future releases of Coq should cut our time back to a few hours, however.)

We compensated in part this performance lag by using more sophisticated algorithms, using multiway decision diagrams (MDDs) [1,12] for the reducibility computation, concrete construction programs for representing configurations, and tree walks over a circular zipper[21] to do configuration matching. This sophistication was in part possible because it was backed by formal verification; we didn't have to "dumb down" computations or recheck their outcome to facilitate reviewing the algorithms, as Robertson *et al.* did for their C programs [27].

Even with the added sophistication, the program verification part was the easiest, most straightforward part of this project. It turned out to be much more difficult to find effective ways of stating and proving "obvious" geometrical properties of planar maps. The approach that succeeded was to turn as many mathematical problems as possible into program verification problems.

4 Prover technology

The scripts of this proof were developed and checked using the Coq v7.3.1 system [13], under the Proof General front end [7]. This combination of tools provided a very flexible basis for trying out different approaches to the problem. In particular, as the proof advanced, we were able to extend the Coq scripting language to directly support the more effective approach that emerged. These extensions enabled progress and ultimately completion of the project, but unfortunately tie down the scripts to the 7.3.1 version of Coq; the user-level syntax extension facilities of the more recent releases of the system, although easier to use, do not support our extensions (and finishing the proof was more of a priority than porting it)[6].

However, the most important feature of the Coq system for this proof was its logic, not its syntax. Specifically, the entire enterprise hinged on the fact that the Coq logical language includes a small but practical programming language, and that the computation of programs embedded in statements is an integral part of the Coq logic. In the Coq logic, the two statements

`P(2 + 2)` and `P(4)`

are not just equivalent, they are *identical*, meaning that the two statements can be used interchangeably at any point in a formal demonstration. As a consequence, one can use any proof of `P(4)` as a proof of `P(2+2)` — e.g., if `P(x)` is defined as `4 = x`, the instance of the reflexivity of = that proves 4 = 4 also proves directly 4 = 2 + 2.

This feature is called *computational reflection*. From the trivial 2+2 example above, one may get the impression that computational reflection is neither sound nor useful:

A. The correctness of demonstration using computational reflection, depends on the irks and quirks of an arcane programming language.
B. If all the feature can do is prove 2 + 2 = 4, there's not much point to it.

Both impressions are utterly wrong. As regards impression A, correctness, the embedded programming language has been carefully trimmed of all the traps and trappings commonly found in general purpose programming languages, to ensure that every expression and statement has a definite, context-independent meaning. The list of features excluded by this principle is long: there are no side effects, since they would make the outcome of a computation depend on its timing; there are no floating point or primitive integers, since their rounding/overflow behavior is machine dependent; consequently there are no constant-time access arrays; there are no exceptions, virtual methods, unbounded loops, … In fact the most complex to implement feature of the language is the ability to call functions with arguments. Since definitions with parameters are the main building block for mathematical theories, and thus any practical formal proof system must include support for replacing parameters with values,

[6] Since this report was written, the proof style pioneered in this work has been implemented by the ssreflect plugin to the Coq system. It is actively maintained, the Four Color proof has been ported to it and is now available on the mathcomp Github repository.

admitting computational reflection requires no great leap of faith. On the contrary, the theory of pure functions with arguments, called the λ-calculus, has been studied for over 75 years. The logical metatheory of Coq, which was established almost 20 years ago, is in fact directly based on the λ-calculus: in Coq, the consistency of the logic follows from the consistency of computations[13,2,8].

Of course, the fact that the embedded language is so Spartan rather strengthens impression B: it seems difficult to write useful code in such a restricted language, despite the well-known fact that in principle, any computation can be described in the pure λ-calculus. We believe our work provides ample evidence that this is definitely not the case, and that on the contrary computational reflection is a very effective proof technique, both in the large and in the small:

- In the large, we use computational reflection to implement major parts of the Four Color Theorem proof, with programs of an unprecedented size and complexity. Most of the previous applications of computational reflection used only relatively modest implementations of formal rewrite systems [11].
- In the small, we use computational reflection to robustly automate many "small steps" of our proof, so we have reasonably short proofs without having to resort to the fickle domain-specific "decision procedures" traditionally used to handle these simple steps.

The flagship application of computational reflection "in the large" occurs in the reducibility part of the proof of the Four Color Theorem. We define a common prologue, in file `cfreducible.v`, that includes the following:

```
Variable cf : config.
Definition check_reducible : bool := …
Definition cfreducible : Prop := …

Lemma check_reducible_valid : check_reducible -> cfreducible.
```

The variable `cf` is a parameter of the definitions and lemmas that follow. It stands for the explicit data representation of a configuration map, basically a string of letters and numbers. We define elsewhere a standard interpretation of that string as an explicit program for constructing a mathematical hypermap object (see section 5.3 for an explanation of that construction). The two definitions that follow are of a very different nature; the first is a program, while the second is a logical predicate:

- `check_reducible` is a Boolean expression that performs a complex combinatorial reducibility check, using a nonstandard interpretation of the map construction program `cf`, and the MDD computations mentioned in Section 3.
- `cfreducible` is a mathematical proposition that asserts that the hypermap constructed by the standard interpretation of `cf` is indeed reducible.

Using these definitions, we then prove the `check_reducible_valid` lemma, which asserts the partial correctness of the `check_reducible` program with respect to the mathematical specification `cfreducible`: if the `check_reducible` returns `true`, then `cfreducible` holds. Since `cf` is a parameter of this proof, the result applies *for any value* of `cf`. Using this, we can prove for example

```
Lemma cfred232 : cfreducible (Config 11 33 37
    H 2 H 13 Y 5 H 10 H 1 H 1 Y 3 H 11 Y 4 H
    9 H 1 Y 3 H 9 Y 6 Y 1 Y 1 Y 3 Y 1 Y Y 1 Y).
```

in just *two* logical steps, by applying `check_reducible_is_valid` to the concrete configuration above and the trivial proof of `true = true`, even though the configuration map represented by `(Config 11 33 …)` has a ring size of 14 and a longhand demonstration would need to go over 20 million cases. Of course, the

complexity does not disappear altogether — Coq 7.3.1 needs an hour to check the validity of this "trivial" proof.

The fact that evaluation is transparent in Coq's logic has another important consequence for us: it allows us to use freely multiple levels of intermediate definitions to derive specific properties from general ones in a modular way. Because such definitions are automatically expanded and specialized by evaluation, we can directly use the lemmas associated with the generic properties on the derived properties, without having to manually unfold the property definitions or provide new specialized lemmas. For example, when in the scripts above we use the Boolean *value* `check_reducible` as a logical statement, Coq silently[7] interprets this as `(is_true check_reducible)` where `(is_true b)` is defined as `b=true`. — this is why we could directly use reflexivity in the proof of `cfred232`.

We use this "trick" extensively in our proofs, because it allows us to handle a wide range of concepts using a relatively modest set of basic lemmas — we never needed to resort to the lemma search command of Coq. For example, we model paths with a Boolean function recursively defined over sequences

```
Variable e : rel d.
Fixpoint path (x : d) (p : seq d) : bool :=
  if p is y :: p' then e x y && path y p' else true.
```

This allows us to directly use the standard lemmas for the Boolean "and" function `andb` to reason about `(path x (y :: p))`, `(path x (y :: z :: q))`, etc.

Using Boolean functions in lieu of logical predicates has other benefits: we can use rewriting of Boolean expressions to apply an equivalence deep inside a complex statement, e.g., to "push" the `path` predicate over a sequence concatenation, using

```
Lemma path_cat (x : d) (p1 p2 : seq d) :
  path x (cat p1 p2) = path x p1 && path (last x p1) p2.
```

Booleans also make it easy to reason by contradiction even in Coq's intuitionistic logic. In particular, if a Boolean assumption computes or rewrites to `false`, then the standard tactic `Discriminate` can be used to conclude the proof. This turned out to be so common that most of our proof tactics attempt to use `discriminate` automatically, and this is so effective that almost all our case splits leave at most two nontrivial subgoals. Thus, we can make the script layout reflect the structure of the proof just by indenting the script for the first subgoal.

Because we employed mostly computable definitions, we were able to rely mainly on a "generate-and-test" style of proof. Most of our proofs follow this routine:

- Do a case split on relevant variables or expressions, based on their type
- Compute out the proof goal, possibly along with some assumptions

Quite often this was enough to complete the (sub)proof; most of remaining cases could be resolved just by nudging the computation with selective rewriting (e.g., using the `path_cat` lemma above). Explicit logical steps came only a distant third (7000 tactic calls, compared to 21,000 stepping/case splitting/rewriting tactic calls); about two thirds of them were backward inference (explicitly forcing a proof schema, spinning off new proof obligations), a third were forward inference (steps that explicitly add new facts). Although there are only 1500 of them, declarative forward steps explicitly

[7] `is_true` is declared as a *coercion* from `bool` to the `Prop` sort.

inserting a subgoal were very important because they helped structure the longer proof scripts.

We found this approach to machine-assisted theorem proving quite effective, enough so that we made very little use of the (admittedly modest) proof automation facilities of the Coq system. This approach provides a very natural way of harnessing the computational resources of the proof assistant: we write and have Coq execute a set of simple, deterministic *programs* that compute with concrete (but partial) *data*. This compares rather favorably with trying to control the assistant with tactics, which are *metaprograms* manipulating *terms*, or with a set of *nondeterministic* rewrite rules.

Of course, this conclusion is relative to the problem at hand—there is not much symbolic algebra in the proof of the Four Color Theorem, and even so there were many subproofs that would have benefited from better rewriting support. Despite these caveats, it remains that our simple-minded approach succeeded on a challenging problem. We should point out that the liberal use of intermediate definitions (over 1000) also greatly helped in keeping the development tractable: there are only 2500 lemmas in the entire proof, more than half of which are trivial rewrite equations for integers and list operations. Unfortunately combining definitions also makes much of the Coq automation ineffective: unless definitions are unfolded just at the right level, the primitive `apply`, `auto`, `autorewrite` and most of the more elaborate tactics will fail to match the goal; user guidance is almost always needed. Moreover, it is unclear that more aggressive unfolding in automated tactics would help, since the size of the goal increases geometrically when several layers of definitions are unfolded.

The size of the individual lemma proofs varies widely: over half of them are one-liners, 75% are under 5 lines, 90% under 10 lines, but at the other end 40 are longer than a screen page, up to almost 700 lines for the proof of correctness of the compilation of configuration reducibility (lemma `cfctr_correct`). We have such large proofs because we deliberately avoid the clutter of ad hoc lemmas that often plague formal developments; if an obscure result is only needed as an intermediate step of a higher-level lemma, we use forward reasoning to introduce it and prove at the point at which it is relevant. This often saves a significant amount of bookkeeping, since we do not need to explicitly establish and introduce the context in which the result holds.

There is a trade-off, however: having longer, more significant lemmas greatly helps bookkeeping at the global proof level, but can create bookkeeping problems for individual proofs. Indeed, the proof context (the set of variables, definitions, and assumptions that can be referenced) can easily become uncomfortably large even in a 10-line script if it is not managed. We found it difficult to do systematic context management, using only the standard set of Coq tactics. For example, it takes two separate commands to discharge an assumption H (moving H from the context to the goal G, so that following case, rewrite or reverse chaining commands will operate on $H \rightarrow G$). Although this may appear to be a minor annoyance, this makes it very difficult to maintain consistency in a script that explicitly moves assumptions to and from the context—but that is largely what context management consists of. It also didn't help that several useful variants of this operation required rather different, sometimes arcane sequences of commands.

This prompted us to develop our own tactic "shell", so that we could make better progress with the project. Fortunately, the Coq v6-v7 systems included a rather powerful, if somewhat unwieldy, user syntax mechanism that was sufficient for our purposes (unfortunately, it has been replaced by an easier to use but less expressive notation facility in recent releases of the Coq system). Although we only put in features when they occurred repeatedly in our scripts, we ended up with a complete and rich language.

We should point out that such extensions have no impact on the trustworthiness of the proof that was carried out, for two reasons:

- There is a small, well-defined *kernel* of the system that checks the logical validity of proofs carried out with the Coq assistant. The bulk of the Coq code base, and in particular the entire tactic and input syntax subsystems can only prepare candidate proofs and submit them to the kernel for validation. For additional safety, Coq actually stores the proofs that have been approved, so that they can be independently rechecked.
- Our extensions only involved modifying the tables of the Coq extensible parser. We did not add any ML code to the system, so our extensions cannot possibly interfere with the kernel code through side effects, language-dependent behavior, or unsafe code.

Our design follows the philosophy of a command shell: we offer a small number of commands, but each of them can perform a variety of related functions, according to its arguments; all commands have identical or at least very similar argument conventions. Nearly 85% of the commands in our scripts are one of the following five[8]:

- `move`: $t_1\ t_2\ t_3\ \ldots$ => $p_1\ p_2\ p_3\ \ldots$.
  This is the basic bookkeeping command, for moving assumptions and values from the context to the goal and conversely. It combines the functions of the primitive Coq commands `generalize` and `intros`, mixed with `clear` and `pattern`. The arguments $t_1\ t_2\ t_3$ are moved to the goal; if t_i is a value rather than a logical assumption, "moving t_i to the goal" implies first generalizing t_i, that is, replacing t_i with a new variable x_i. After the =>, $p_1\ p_2\ p_3$ provide names for the assumptions or values that are moved from the goal to context; $p_1\ p_2\ p_3$ can actually be Coq "intro patterns", allowing tuples, lists, or conjunctions to be decomposed on the fly. The `move` command has several other options for deleting additional assumptions, selecting the exact occurrences to generalize, and performing on-the-fly simplification.
- `case`: $t_1\ t_2\ t_3\ \ldots$ => [p_{11} | p_{12} | …] $p_2\ p_3\ \ldots$.
  The `case` command is a version of `move` specialized for the analysis of data structures: the first pattern to the right of => must be a destructuring "intro pattern" [13]. The `case` command provides additional facilities for dealing with Coq's dependent datatypes, but is otherwise interchangeable with the `move` command.
- `apply`: $P\ldots$ => [p_{11} | p_{12} | …] ….

[8] We reuse the names of some of the primitive Coq commands, but ensure backward compatibility by always including a symbol (one of : / =>) in the list of arguments. The original tactic shell used the Coq 7.3.1 tactic names, but this report has been revised to use the names used by the current ssreflect plugin.

This is the basic reverse chaining command; it is similar to Coq's primitive `apply`, with two important differences: it is insensitive to intermediate definitions, and the proof method *P* can contain holes "`?`" that will be filled by matching the current proof goal. The other command arguments are similar to those of the `Case` command, except that the patterns p_{11} , p_{12} are used on the first, second, etc. subgoal, respectively.

- `have` *p*`:` *t* `by` …
  This is the forward chaining command, similar to Coq's primitive `assert`, except that it can be used to introduce several values (*p* can be a pattern), and the `by` introduces a tactic, not a term. The `by` can be omitted, in which case the command generates a subgoal for the justification of *t* (only 25% of the `have` commands have a `by` in the Four Color proof script).
- `rewrite` t_1 t_2 $t_{3...}$
  This command is the main workhorse in our scripts; it is called nearly 10000 times, almost as many times as `move` and `case` combined. It merges the functionality of most of Coq's rewriting/computing commands (`rewrite`, `unfold`, `fold`, `and simpl`) with a coherent, unified notation, also adding important functionality such as selection of the rewrite occurrences. Most importantly, however, it applies a *sequence* of rewrite rules rather than a single rule; this feature alone about halves the size of our scripts!

These commands and a handful of related ones (`elim`, `clear`, `and congr`) are defined and extensively documented in the prelude file `tacticext.v` that is loaded by every file in the development. The `move`, `case` and `apply` commands also have a "view" feature that provides a tie-in for the `reflect` inductive predicate defined in the Boolean prelude file `ssrbool.v`. Briefly, `(reflect` *P b*`)` states that *P* and *b* are equivalent (i.e., *P* ↔ *b*`=true`). We prove such properties for all Boolean predicates that have a useful logical counterpart; for a predicate `foob`, the corresponding lemma is `fooP`. For example in `ssrbool.v` we prove

`Lemma andP :` `reflect` (b_1 `/\` b_2) (b_1 `&&` b_2)`.`

Then we can write `move/andP=> [Hb1 Hb2]` to decompose an assumption of the form b_1 `&&` b_2 as if it were b_1 `/\` b_2, that is, introducing assumptions `Hb1: b1` and `Hb2 : b2`. Conversely, we can use the sequence `apply/andP; split` to split a proof of b_1 `&&` b_2 in two subproofs of b_1 and b_2, respectively.

Since we adopted a "command shell" design, our scripts do not read as mathematical text. However, given our proof style based on computation and expansion, we don't think they possibly could, as we tend to get many large, unwieldy subgoals. The important thing for us was that this language allowed us to tackle effectively complex proofs, even when we didn't have a detailed text proof to guide us. We can present this piece of anecdotic evidence as to the efficiency of our command language: it turned out we needed a variant of Streicher's axiom K [19], which states that there is only one proof of x = x, for our `eqType` structures. Axiom K is derivable in this case, because equality is decidable by definition on `eqType`; in fact this derivation is in the Coq standard library (in `Logic/Eqdep_dec.v`). We tried rolling out our own proof, and came up with a four-line, 50-word proof script; the standard library proof has over 100 non-comment source lines, and 500 words, so we can claim an order-of-magnitude productivity improvement on this particular example.

5 The computer-checked proof

The exercise of formalizing a famous theorem can easily degenerate in the following (somewhat depressing) routine: translate a nice, crisp, mathematical text into computer lingo, at the rate of xx days per page, balk at the long list of trivial mathematical prerequisites that ought to be known to the proof system (but aren't), write a large ad hoc collection of lemmas to fill the deficiencies.

However, our experience with the Four Color Theorem wasn't like this. Because we were faced with rather different implementation and performance constraints, we had to come up with our own programs for checking reducibility and unavoidability (although we did reuse as much of the Robertson *et al.* proof as we could). Because most graph theory proofs rely on the visual analysis faculties of the reader, and a proof assistant like Coq is fully devoid of such faculties, we mostly had to come up with our own proofs (sometimes using visual intuition as guidance, but not always).

In addition, as we noted in section 2, we were working with a different, more combinatorial notion of map; while this notion was generally more cumbersome than the more familiar topological one, on several occasions its additional precision provided a clean argument where the "intuitive" situation seemed muddled at best. In particular we found we could do without the notion of "ring completion" and the "omitted straightforward but lengthy proof" of the "folklore theorem (**3.3**)" in the Robertson *et al.* paper [26]. Indeed, the biggest payoff was that we could weaken the definition of "appears", which involves graph *isomorphism*, into one that involved only *homomorphism*, thereby simplifying considerably the configuration occurrence check and its correctness proof.

The remainder of this section gives an account of the mathematics and the algorithms involved in the actual computer proof: the theory of planar hypermaps, the reducibility computation, the construction programs for configuration maps and their interpretations, the Birkhoff Lemma and Theorem and the embedding of configurations in a minimal counter example, the enumeration of second neighborhoods, and the discretization of the continuous Four Color problem to the hypermap one.

5.1 Hypermaps

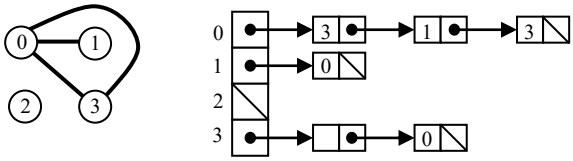
Having ruled out topological definitions of planar maps, we needed a combinatorial replacement for them in order to state (and hopefully prove) the Four Color Theorem. That structure would have to

a) explicitly represent all local geometrical connections
b) support a clearly recognizable definition of "planar"
c) be easy to manipulate in Coq

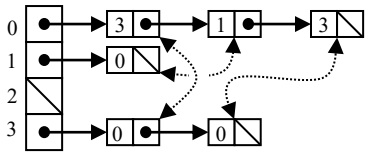
This combination of requirements ruled out all the structures that had been used in the previous formal developments of graph theory, e.g., simplicial representations, which have separate sorts for edges, faces, and nodes are too complex to meet c), and graphs verifying the Kuratowski minor exclusion [23] fail both a) and c). Rather than combing the literature in search of an appropriate definition, we tried to roll out our own from first principles, and ended up rediscovering hypermaps, which are well-known combinatorial structures [15,31,32,33,29,16].

We reasoned as follows:

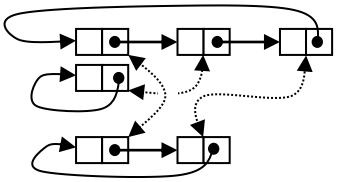
1. Since an essential part of our approach was to define properties by computable programs, we wanted a representation on which graph traversal would be easy to program. A natural starting point was therefore the basic data structure for directed graphs, an array G of lists of integers, where $G[i]$ lists the immediate neighbors of node i (nodes are represented by integer indices).



2. Each cell in these lists corresponds to an edge in the directed graph. Additional edge attributes, such as labels, are often added to the cells; in particular, for symmetric graphs, it is common to add reciprocal pointers between the two cells that correspond to the same undirected edge in the graph. More precisely, when adding an edge $e = \{i,j\}$ to a symmetric graph, that is, adding j to $G[i]$ and i to $G[j]$, means inserting a cell c_j in $G[i]$ and a cell c_i in $G[j]$, *and* putting extra pointers $c_j \rightarrow c_i$ and $c_i \rightarrow c_j$ in c_i and c_j, respectively. In this way, the edge e is represented by the linked pair $c_i \leftrightarrow c_j$: each of the cells c_i, c_j represents a "half-edge", called a *dart*, in the hypermap terminology (some authors use the term *flag*).



3. The structure can encode geometrical information at no extra cost, simply by taking the convention that $G[i]$ list the darts of node i in geometrical order (say, counter clockwise). Since this is a cyclic order, it is then natural to make the cell lists circular, by having the last cell point to the first. This simplifies considerably geometrical traversals: to go from any dart to the next dart on the same face, *clockwise*, simply follow, successively, the "edge reciprocal" and "node list" pointers. Moreover, each non-isolated node of the graph is now represented by its circular list, just as the edges are represented by the reciprocal dart pairs. This allows us to further simplify the structure by removing the G array and the integer indices altogether, as the "edge" pointer of a cell c_j always points into the same list as $G[j]$. We assume that we also have some way of iterating over all cells.



4. The resulting structure is easy to represent in Coq: there is only one type of objects, d (for darts), whose only properties is that it is has an equality test and is finite. Pointer attributes like the "edge" and "node" pointers above become just functions $e, n : d \rightarrow d$, and in general any additional attribute (e.g., "color") can be represented by a function on d. Here n and e must be, respectively, a permutation and an involution on d. Weakening the requirement for e to just being a permutation, that is, allowing "edges" to have more than two sides, yields the traditional definition of

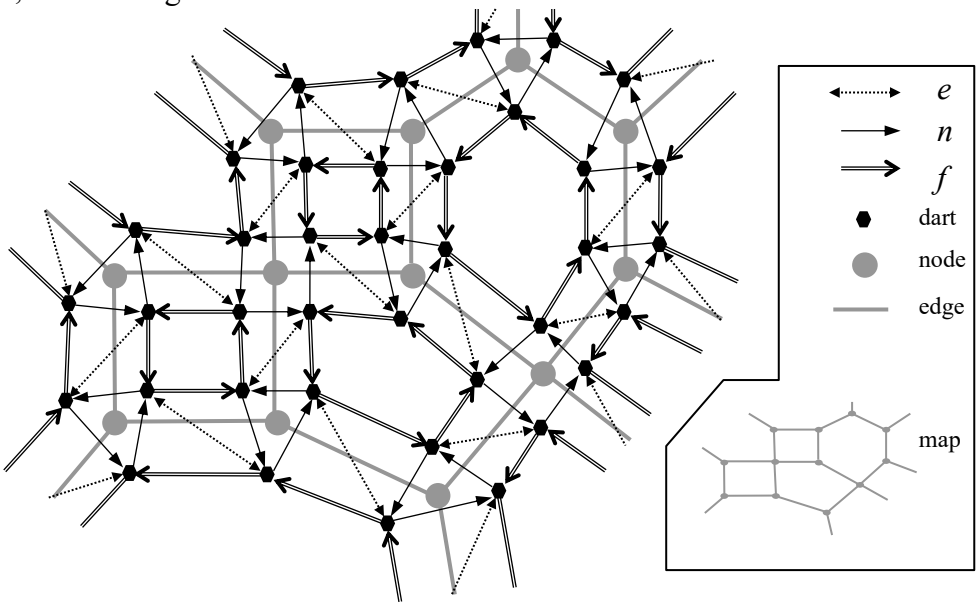
hypermaps: a pair of permutations of a finite set. Hypermaps have been extensively used in the study of graph enumeration [16].

5. However, even this simplified definition turned out to be impractical in Coq, because "being a permutation" property is moderately difficult to prove and use: both the "there exists an inverse" and the "injective" characterizations require *deductive* steps to use; in our approach, this is five to ten times more expensive than an equational characterization, because a single `rewrite` command can perform five to ten rewrite steps. For this reason, we extended the structure with a third dart attribute, a pointer to the next *counter clockwise* dart on the same face, represented by a function $f : d \to d$. The navigation trick pointed out in 3. above turns into an *equation*

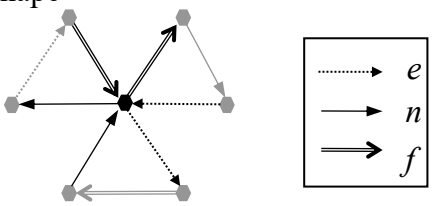
$$e \circ n \circ f = \mathrm{Id}$$

Because d is finite, this identity *characterizes* hypermaps; it implies that e, n, and f are all permutations, and that we also have $n \circ f \circ e = \mathrm{Id}$ and $f \circ e \circ n = \mathrm{Id}$. As Tutte pointed out [31], this representation is circularly symmetrical with respect to edges, nodes, and faces.

6. Although their name suggests drawing "darts" as arrows on the original map, this leads to horrendously confused figures when one adds arrows for the e, n, and f functions. In the figures below, we therefore always depict darts as points (we use small hexagonal bullets). When we draw an ordinary map together with the corresponding hypermap, we arrange the darts in circles around the corresponding nodes, halfway between adjacent edges. This way, each node appears at the center of the corresponding n cycle, each f cycle is inset in the corresponding face, and each e cycle is the diagonal of an n-f quadrilateral centered on the corresponding edge, as in the figure below.



However, we rarely find it necessary to draw such complex figures; most of the time, we only need to consider the diagram around a single dart, which has the following fixed, regular shape



The hypermaps that correspond to plain maps have the property that all e cycles have length 2; this implies the additional identities $e = e^{-1} = n \circ f$. We call such hypermaps *plain* hypermaps. Although the central part of the proof of the Four Color Theorem uses only plain hypermaps, most of the basic results on hypermaps are easier to establish for general hypermaps: many of the constructions described in this section create general hypermaps.

Although we primarily selected the "one domain, three functions, one equation" definition of hypermaps for its simplicity, its threefold symmetry allows significant reuse of definitions and lemmas. For example, the Euler formula takes a completely symmetrical form for this presentation of hypermaps

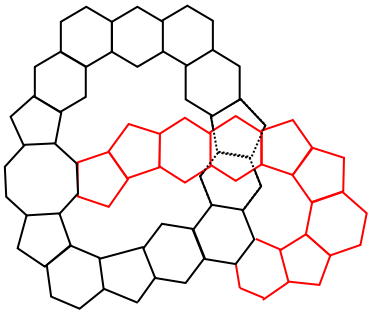
$$\#e + \#n + \#f = \#d + 2\#(e \cup n \cup f)$$

where $\#e$, $\#n$ and $\#f$ denote the number of distinct cycles of e, n, and f, respectively, #d the total number of darts, and $\#(e \cup n \cup f)$ denotes the number of connected components of the relation obtained by taking the union of the graphs of all three functions. Because of this symmetry, the graph/map duality is completely trivial in our setting, so there is no point for us in switching from map coloring to graph coloring, as is traditionally done; all the lemmas in our development are therefore phrased in terms of map coloring.

We actually chose to *define* planar hypermaps as those that satisfy this generalized Euler formula, since this property is readily computable. Much of the Four Color Theorem proof can be carried out using only this formula. In particular Benjamin Werner, who worked with us on the early part of the proof, found out that the proof of correctness of the reducibility part (step I) was most naturally carried out using the Euler formula only. As the other unavoidability part of the proof (steps II) is explicitly based on the Euler formula, one could be misled into thinking that the whole theorem is a direct consequence of the Euler formula. This is not the case, however, because unavoidability also depends on the Birkhoff Theorem giving the shape of second neighborhoods. Part of the proof of the latter requires cutting out the submap inside an arbitrary simple ring of 2 to 5 faces in the (hypothesized) counterexample map. Identifying the inside of a ring is exactly what the Jordan Curve Theorem is about, so this calls for a combinatorial analogue of that famous theorem. As a side benefit, we will get additional assurance that our definition of planarity is correct by proving that the hypermap Jordan property is actually equivalent to the hypermap Euler formula (about half of that converse proof is a lemma required for the correctness of the reducibility check).

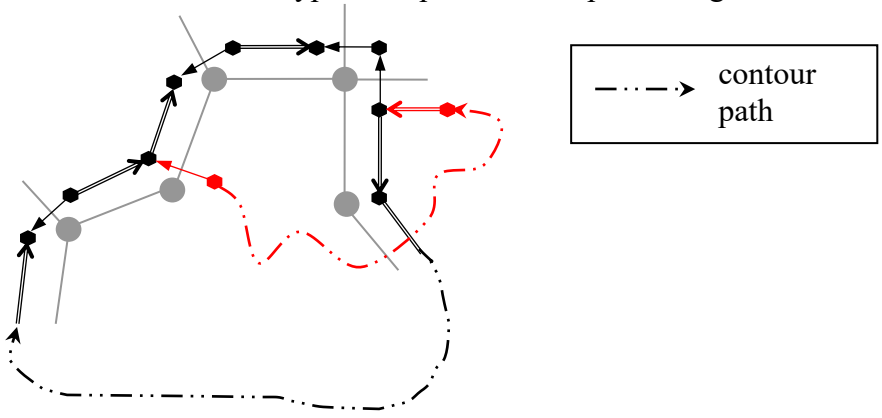
Unfortunately, the naïve transposition of the Jordan Curve Theorem from the continuous plane to discrete maps fails. Simply removing a ring from a hypermap, even a connected one, can leave behind any number of components: both the "inside" and the "outside" may turn out to be empty or disconnected. A possible solution, proposed by Stahl [29], is to consider paths that go from one face of the ring to another (loops are allowed). The Jordan Curve Theorem then tells us that such paths cannot start from the "inner half" of a ring face, and end at the "outer half" of a ring face,[9] i.e., there are no "Moebius rings" such as the one below.

[9] More precisely, this follows from the Jordan separation theorem; the nonseparation part of the theorem, which implies that each edge borders at most two regions, is implicitly used to turn the problem of coloring the continuous plane into one of coloring a discrete data structure.



Given careful definitions of the "inner" and "outer" halves of ring faces, this can be made to work (indeed, we have done this in file `rjordan.v`, as an addendum to the proof), but the resulting definitions are too unwieldy and complex to be practical for machine-checked inductive proofs. Fortunately, a much simpler and slightly stronger version of the Jordan property can be defined *directly* on the hypermap structure, by considering contours of the hypermap three-function graph, rather than rings in an underlying map.

The simplification comes from the fixed local structure of hypermaps, which allows "inner" and "outer" to be defined locally, provided we restrict ourselves to a certain pattern of traversal. Specifically, we exclude one of the three functions (the e permutation), and pick *fixed*, *opposite* directions of travel on the other two: from a dart x we can go to either $n^{-1}(x)$ or $f(x)$. We define *contour* paths as dart paths that only take steps in this $n^{-1} \cup f$ relation. A contour cycle follows the inside border of a face ring, *clockwise*, listing explicitly all the darts in this border. Note that n or n^{-1} steps from a contour cycle always go inside the contour, while f or f^{-1} steps always go outside. Therefore, the Jordan property for hypermap contours is: "no contour path that starts and ends on a duplicate-free contour cycle, without otherwise intersecting it, can start with an n^{-1} step and end with an f step", or, more symmetrically, "any such path must start and end with the same type of step". For example, the figure below is forbidden.

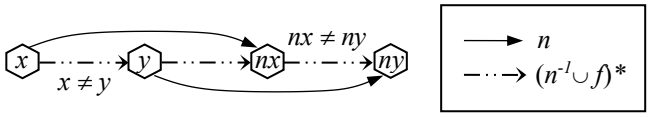


This definition is much simpler than the ring one, not only because it avoids the delicate definition of the inner and outer "halves" of a face cycle, but also because "nonintersecting" (resp., "duplicate-free") have their literal meanings here: not containing the same dart (resp., twice the same dart), whereas for rings we need to also exclude darts in the same f cycle. This also implies that the contour-based property is slightly stronger than the ring property, since it applies to contour cycles that are not derived from simple rings (different darts from the same f cycle can occur in two different sections of a contour cycle). Such contours actually occur in the Four Color

Theorem proof, but only around explicitly matched configuration maps, for which we do not need the Jordan property.

Although it is much simpler than the ring property, we found that the definition above was still too complex to carry out inductive proofs in Coq; it still involves three different values (the path, its starting point, and the cycle), along with five to six separate properties. However, we observed that by splicing together the cycle and the path, we could form an equivalent statement that only involves a single value and three properties. We therefore used the following:

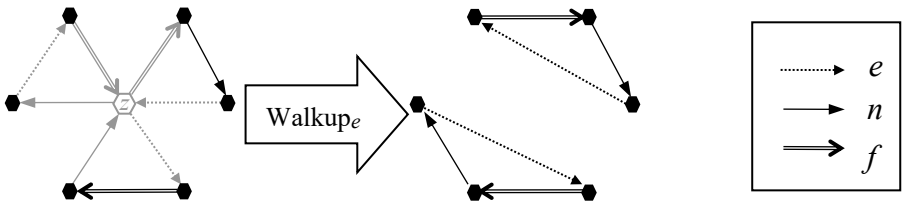
> **Theorem (the Jordan Curve Theorem for hypermaps):** *A hypermap is planar if and only if it has no duplicate-free "Moebius contours" of the form*



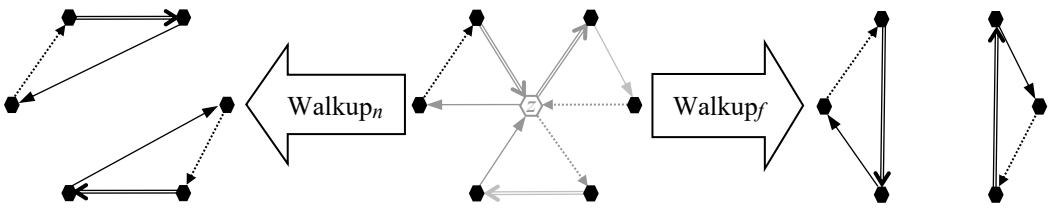
The $x \neq y$ condition rules out contour cycles; note however that we do allow $y = n(x)$.

As far as we know this is a new combinatorial definition of planarity. Perhaps it has escaped attention because a crucial detail, reversing one of the permutations, is obscured for plain maps (where $e^{-1} = e$), or when considering only the cycles of that permutation. Since, as we show in the Coq development, this Jordan property is equivalent to the Euler identity, so it is symmetrical with respect to the choice of the two permutations that define "contours", despite appearances (we use this in the reducibility proof). Oddly enough, we know no simple direct proof of this fact (the best we can do is to derive it from the equivalence between the "ring" and "contour" versions of the Jordan property, and which is hardly "simple").

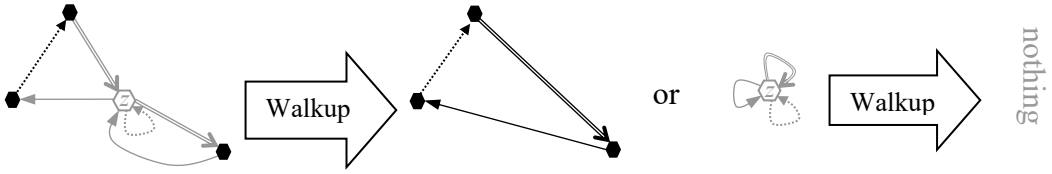
We show that our Jordan property is equivalent to the Euler identity by induction on the number of darts. At each induction step we remove some dart z from the hypermap structure. In doing so, we must take care to redefine the permutations so that they avoid z. For two of them we can do this in the obvious way by suppressing z from the cycle in which it occurs: for example, we can define permutations n' and f' on the smaller hypermap by $n'(n^{-1}(z)) = n(z), f'(f^{-1}(z)) = f(z)$, and $n'(x) = n(x)$ and $f'(x)=f(x)$ otherwise. For the third permutation, however, the triangular identity of hypermaps leaves us no choice, and we have to either merge two cycles, or split a cycle of that permutation. For example, if that third permutation is e, we do the following transformation on the portion of hypermap surrounding z:



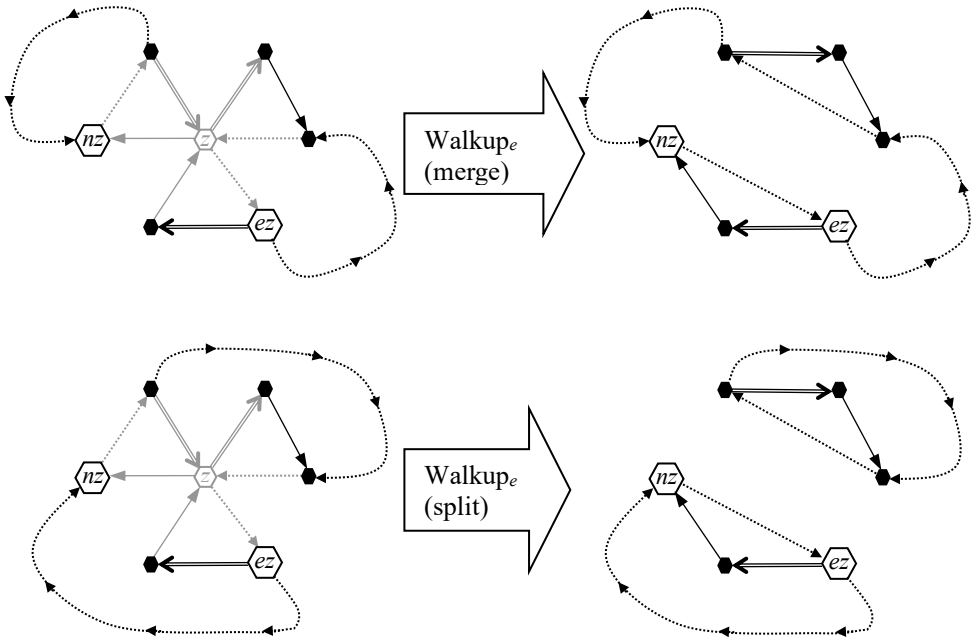
Following Stahl [29], we call this operation the *Walkup* transformation [34]. More precisely, the figure above illustrates the Walkup_e transformation; by symmetry, we also have Walkup_n and Walkup_f transformations.



In general, the three transformations yield different hypermaps, and all three prove to be useful. However, in the degenerate case where z is a fix point of any one of the three permutations, then all three transformations give the same result (z has at most 3 neighbours to connect), e.g., if $e(z) = z$, then either



Of course, we only need to define and study one of the Walkup transformations, as we then get the others (and their properties) for free by symmetry. We chose the Walkup$_e$ transformation. Apart from for the removal of z, the Walkup$_e$ transformation leaves the cycles of n and f unchanged; however, except in the degenerate cases above, it has a nontrivial effect on the cycles of e: if z and $n(z)$ are on different e cycles, the Walkup$_e$ transformation *merges* the two cycles; if z and $n(z)$ are on the same e cycle, then the Walkup$_e$ transformation *splits* this cycle.



The degenerate and merge forms of the Walkup transformation clearly leave the validity of the hypermap Euler equation

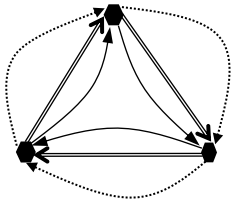
$$\#e + \#n + \#f = \#d + 2\#(e \cup n \cup f)$$

unchanged (both sides decrease by 1 or 3). The split form only preserves the Euler equation if it *disconnects* the hypermap; otherwise, it increases the difference between the left- and right-hand sides by 2. Since repeatedly applying the transformation

eventually yields the empty hypermap, for which the Euler equation is trivially valid, we immediately see that all hypermaps satisfy the inequality

$$\#e + \#n + \#f \leq \#d + 2\#(e \cup n \cup f)$$

Planar hypermaps are thus those that maximize the left-hand side, so applying any of the Walkup transformations to a planar map always yields a planar map; in the split case, the map is always disconnected (as is obvious from the figure above). Thus, to prove the implication Euler→Jordan we are free to apply any sequence of Walkup transformations to reduce a planar hypermap containing a Moebius contour to the map below, for which the Euler equality obviously fails.



We use Walkup$_e$ to eliminate darts outside the contour, as this leaves the n^{-1} and f steps of the contour unchanged, and Walkup$_f$ and Walkup$_n$ to contract n^{-1} and f steps on the contour, respectively. In the Jordan → Euler direction, we use only Walkup$_e$ transformations, as they leave the contour steps mostly unchanged. We carefully select the removed dart to make sure that we are *not* in the split case: we use the Jordan property to show that any e cycle C that is closed under n ($n(z) \in C$ for all $z \in C$) contains a fix point of either n or f. We named the latter statement the Euler tree lemma, because it is the hypermap analogue of the following: if a planar connected graph has only one face, then it is a tree. This property is used implicitly in the traditional "flooding" proof of the Euler formula.

We also use all three transformations in the main body of the Four Color Theorem proof. Since at this point we are restricting ourselves to *plain* maps (all e cycles have length 2), we always perform two Walkup transformations in succession; the first one always has the merge form, the second one is always degenerate, and always yields a plain map. Each variant of this *double Walkup* transformation has a different geometric interpretation, and is used in a different part of the proof:

- The double Walkup$_f$ transformation *erases* an edge in the map, merging the two adjoining faces. It is used in the main induction of the Four Color Theorem proof, to replace an identified reducible configuration with a smaller submap.
- The double Walkup$_e$ transformation *concatenates* two successive edges in the map; we only apply it at nodes that have only two incident edges, to remove *edge subdivisions* left over after erasing edges (the node degree condition ensures that the resulting map is plain).
- The double Walkup$_n$ transformation *contracts* an edge in the map, merging its two endpoints together. It is used to prove the correctness of the reducibility check, by induction on the size of the remainder of the graph. We use the upper cited Euler tree lemma of the Jordan → Euler proof to find an appropriate dart at which to apply the transformation, so as to avoid the split case of the Walkup transformation.

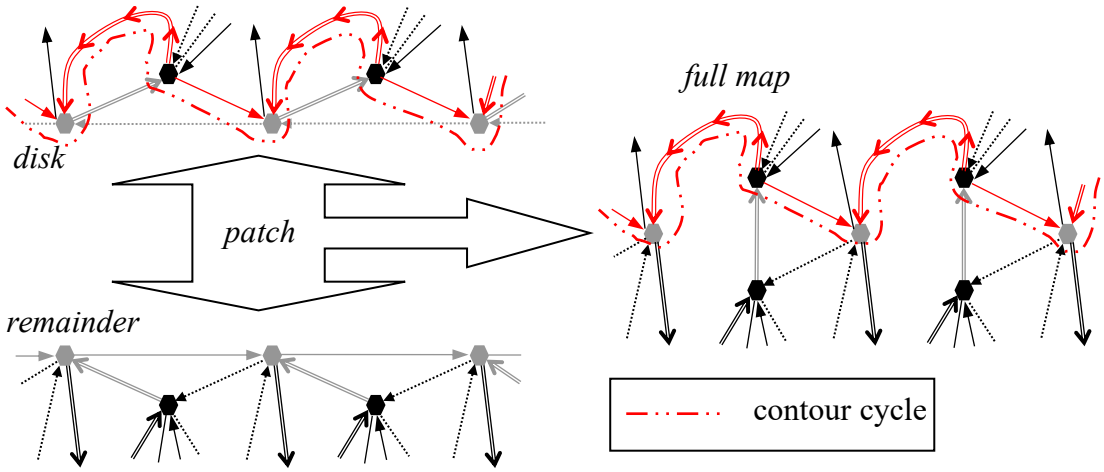
Contours play a central part in our formal development of the Four Color Theorem proof, because they are the basis for a precise definition of the *patch* operation, which pastes two maps along a border ring to generate a larger map. This operation is the cornerstone of the proof, since it defines the three-way relation between a configuration

submap, whether explicitly identified or computed using the Jordan property, the map in which it occurs, and the remainder of that map. Mathematical articles and books rarely define this operation precisely, probably on the account that it is "intuitively obvious". We have found, however, that getting the details of this operation exactly right was critical to the successful proof of several key lemmas. Ultimately, this attention to detail paid off handsomely, as it allowed us to remove injectivity checks in the matching configuration, and to omit the analysis of the structure of the second neighborhood altogether.

A key observation is that, despite appearances, the patch operation is *not* symmetrical in its arguments. It does paste two hypermaps along a ring, but in a slightly asymmetrical manner, and requires different assumptions the two maps:

- For one of the submaps, which we shall call the *disk* map, the ring is an *e* cycle, that is, a hyperedge. This cycle must be *simple*, i.e., no two darts on it can belong to the same face.
- The other submap, the *remainder* map, the ring is an arbitrary *n* cycle.

The gluing operation itself consists in merging pairwise the darts on the *e* cycle of the disk map with those on the *n* cycle of the remainder map, *reversed* (the two cycles must have the same length). On the merged cycle, the *e* function is defined as in the remainder map and the *n* function is defined as in the disk map. The definition of the *f* function is then adjusted as is to satisfy the triangular identity, as illustrated below.

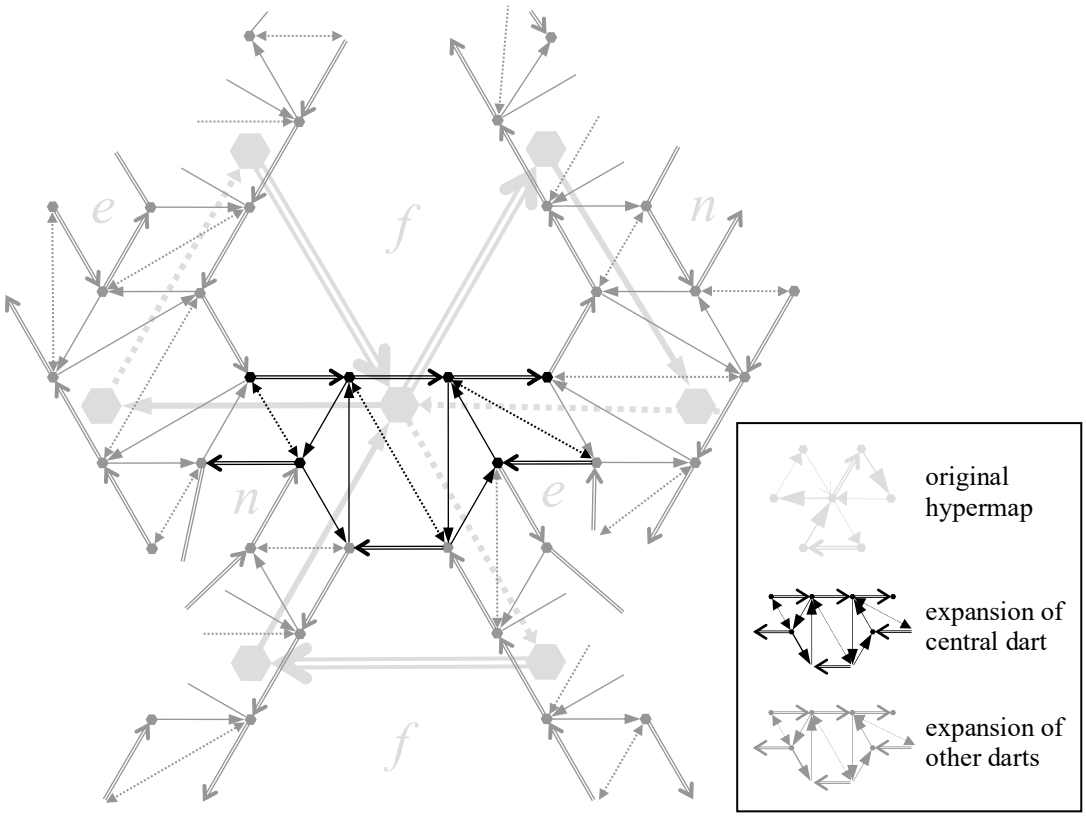


Let us point out that although the darts on the border rings were linked by the *e* and *n* permutations in the disk and remainder map, respectively, they are not directly connected in the full map. Indeed, they need not even form a simple ring in the full map. However, because the *e* cycle is simple in the disk map, it is a sub cycle of a *contour* that delineates the entire disk map (indicated by the mixed dash in the figure above). This contour *is* preserved by the construction, which, because of this, is reversible: the disk map can be extracted, using the Jordan property, from this contour. This allows us to divide the proof load by two, using equational reasoning to show that geometrical properties in the full map are *equivalent* to the conjunction of similar properties in the disk and remainder maps:

- The full map is planar (resp., connected) if and only if both submaps are.
- The full map is plain if and only if the remainder map is plain, and the disk map is plain except for the border ring.
- The full map is cubic (all *n* cycles of length 3) if and only if the disk map is cubic, and the remainder map is cubic except for the border ring.

- If the full map is bridgeless both submaps are; the converse only holds if the disk map ring is *chordless* (where a *chord* is an *e* link between the *f* cycles of nonconsecutive darts in the *e* cycle ring).
- The full map is four colorable if and only if both submaps are four colorable with colorings that match on the rings (with a reversal).

All the theory exposed so far applies to general hypermaps. However, for the rest of the proof of the Four Color Theorem we need to specialize to plain (all *e* cycles of length 2) and cubic (all *n* cycles of length 3) maps: these properties are required both for the enumeration strategy in the unavoidability part, as well as for the computation of reducibility checks. To do this specialization we show that the task of coloring an arbitrary hypermap can be reduced to coloring the hypermap obtained by covering every node and hyperedge with a new face. As depicted in the figure below, the construction of this new map is completely regular and straightforward; the new map has six times as many darts as the original one.



5.2 Reducibility

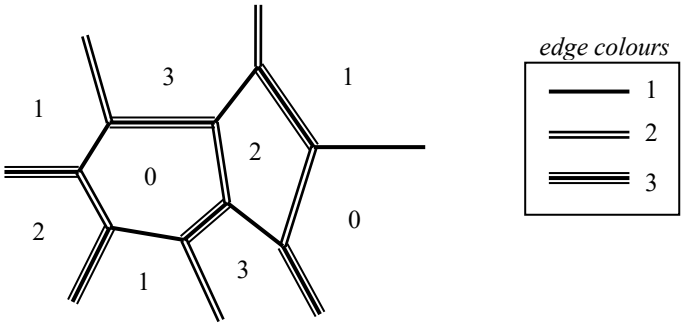
The bulk of the reducibility computation for a given configuration consists in iterating a formalized version of the Kempe chain argument, in order to compute a lower bound for the set of colorings that can be "fitted" to match a coloring of the configuration border, using color swaps. The actual check consists in verifying that this lower bound contains all the colorings of the map obtained by erasing 1 to 4 specific edges in the complete configuration map; this smaller map is called the *contract map* of the configuration. This final check does not require a significant amount of computation (in many cases, the lower bound yields the set of all possible colorings, whence the check is trivial).

Although the Kempe chain argument is usually formulated for the dual graph coloring problem, we have found that the form that is actually used for the proof of the Four Color Theorem is easier to justify for the map coloring problem. The argument uses an alternative formulation of the coloring problem, proposed by Tait[30] in 1880 (one year after Kempe's original proof, along with a variant of Kempe's proof that turned out to be equally wrong).

Tait suggested replacing the face coloring problem with an edge coloring problem. Suppose we use the integers 0, 1, 2, and 3 to "color" the map faces; given such a coloring k, we can compute another coloring, this time on the edges of the map: we color an edge x that separates two faces a and b with the bitwise sum (aka. exclusive or) $k(x) = k(a) \oplus k(b)$ of the colors of a and b. This edge coloring has two obvious properties:

1) Since a and b are adjacent we have $k(a) \neq k(b)$, so $k(x)$ takes only the values 1, 2, or 3
2) The bitwise sum of $k(y)$ for all edges y that are incident to any given node p is 0: expanding the definition of $k(y)$, we see that the color of every face incident to p occurs twice in the summation.

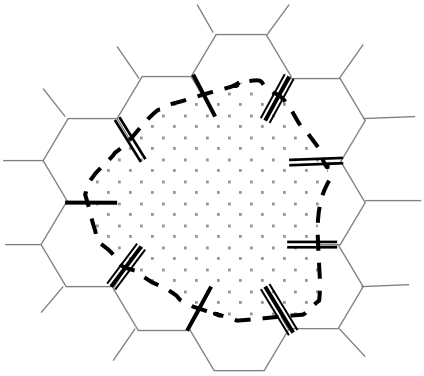
Conversely, any edge coloring that satisfies 1) and 2) can be turned into a face coloring, by choosing an arbitrary color for one face a_0 in each connected component of the map, for each edge x bounding a_0, coloring the face across x with $k(a_0) \oplus k(x)$, and so on (the formal proof of correctness involves the Jordan property).



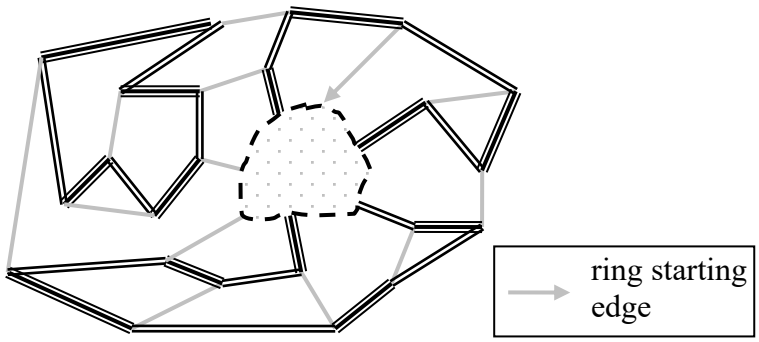
Tait further observed that for *cubic* maps, 2) is equivalent to the much simpler

3) For any node p the colors of the three edges incident to p are a permutation of {1,2,3}.

He concluded that four coloring the faces of a cubic planar is equivalent to three coloring its edges. Now, consider the problem of trying to fit the coloring of a submap with a coloring of the remainder of the map, where the submap has been "cut out" across a ring of faces. By Tait's results, we only need to match the colors of the edges that were cut.



This simple observation has a significant impact on complexity, since we now have only three colors to consider, and there are only six permutations of them. There is more, however. Consider an arbitrary edge coloring of the remainder part of the map, and erase all the edges with the color 1. Since this deletes exactly one of the three edges incident to each node, the resulting map is the union of a set of isolated loops with a set of linear paths ("chords") starting and ending with the 2- and 3-colored edges dangling at the cut-out (see the figure below). Since we are only interested in the coloring of these dangling edges, we shall ignore the loops.



The color of edges on any of the chords must strictly alternate between 2 and 3, so the dangling extremities of a chord must have the same color if the length of the chord is odd, and different colors if the length of the chord is even. In fact, this is the *only* constraint on the coloring of chords, since exchanging colors 2 and 3 on any one chord yields a valid edge coloring of the remainder. Since we can do this *chord flip* simultaneously for any subset of the chords, we can match any edge coloring of the cut-out submap that

a) assigns color 1 to the same dangling edges as the remainder coloring
b) assigns different colors to the ends of a chord if and only if the length of the chord is even.

We say that such a coloring is *consistent* with the set of chords.

Let us say that an edge coloring of the remainder map is *suitable* if it can be transformed, through a sequence alternating the chord flips described above with global permutations of the three edge colors, into an edge coloring that matches exactly an edge coloring of the cut-out submap on the sequence of dangling edges. We prove the reducibility of the cut-out map by showing that *every* coloring of the remainder map that matches the coloring of a specific, strictly smaller, *contract map*, is suitable. This implies that it is impossible for a minimal non four colorable map to contain the cut-out, since by minimality the map obtained by replacing the cut-out with the contract would have a valid edge coloring, whose restriction to the remainder would then be suitable.

The bulk of the reducibility check consists therefore in constructing a safe approximation of the set Θ of the sequences of colors assigned to the dangling edges by suitable colorings of the remainder, using a fix point computation. We compute an increasing sequence $\Theta_1, \Theta_2, \Theta_3 \ldots$ of safe approximations of Θ until it converges:

i. In Θ_1 we only put the sequences of colors assigned to the dangling edges by actual colorings of the cut-out.
ii. If $\theta \in \Theta_i$, we can add $\rho\theta$ to Θ_{i+1}, where ρ is any permutation of $\{1,2,3\}$.

iii. By the above, we could also add to Θ_{i+1} a sequence θ of colors if we could find some $\theta' \in \Theta_i$ and set of chords ξ induced by an edge coloring of the remainder such that both θ and θ' are consistent with ξ. We can't really do this, since we don't know the remainder. However, since any θ induced by an edge coloring of must be consistent with *some* set ξ of chords, if we can find a θ' as above for *all* ξ that θ is consistent with, we can safely add θ to Θ_{i+1}.

Step iii is practical because the conditions a) and b) above only depend on the endpoints and the parity of chords, and not on their internal edges. This endpoint/parity information can be very simply represented by a formal word in a bracket (or Dyck) language, as follows:

For each dangling edge, in counter clockwise order, starting from any fixed edge:

- Write down a dash '-' if the edge is not part of a chord.
- Write down an open bracket '[' if the edge is the start of a chord.
- Write down a close bracket '$]_b$' if the edge is the end of a chord with "parity" $b \in \{0,1\}$, with $b = 0$ if and only if the chord has an *odd* length.

Since the remainder map is planar (more precisely, outerplanar), chords cannot cross; therefore, if we read back the codeword while going around the dangling edges, we can unambiguously interpret a closing bracket as marking the end of the last chord for which we have seen the start but not the end (which we'll refer to as an *open chord*). We call these four-letter codewords *chromograms*. For example the chromogram for the last figure, starting with the edge marked by an arrow, is $\langle$- [- - [ ]$_1$]$_0$ $\rangle$.

The "consistent with" relation between edge colorings and chromograms can be defined with a simple formal recurrence. Let us write

- ε for both the null color sequence and the null chromogram
- $\theta \approx \gamma$ for "the sequence of edge colors θ is consistent with the chromogram γ"
- $\nu / \theta \approx \gamma$ for "the sequence of edge colors θ is consistent with the chromogram γ, given that the sequence of edge colors of the beginning of open chords is ν"

We then have

- $\theta \approx \gamma$ iff $\varepsilon / \theta \approx \gamma$
- $\nu / 1\theta \approx \text{-}\,\gamma$ iff $\nu / \theta \approx \gamma$
- $\nu / c\theta \approx [\,\gamma$ iff $c \in \{2,3\}$ and $\nu c / \theta \approx \gamma$
- $\nu c / c'\theta \approx]_b\,\gamma$ iff $c' = c \oplus b$ and $\nu / \theta \approx \gamma$
- $\varepsilon / \varepsilon \approx \varepsilon$

Writing $[\theta]_{\approx} = \{\gamma : \theta \approx \gamma\}$ for the set of chromograms that a color sequence is consistent with, we can reformulate step iii. of the reducibility computation as

iv. Compute $\Gamma_i = \bigcup_{\theta \in \Theta_i} \{\gamma \mid \theta \approx \gamma\}$

v. Add to Θ_{i+1} any θ for which $\{\gamma \mid \theta \approx \gamma\} \subseteq \Gamma_i$

The time and memory required for the fix point computation increases geometrically with the ring size of the cut-out. For the largest ring size (14) there are about 300,000 different valid edge color sequences, about 1,000,000 chromograms, and around 20,000,000 color sequence/chromogram pairs in the '≈' relation. While these numbers are by no means out of reach of modern computers, we need to be somewhat cautious, since the computation needs to run inside the proof checking kernel of Coq, which as a compute engine is several orders of magnitude less efficient than the bare processor. Also, we must select an algorithm that does not use imperative structures such as arrays,

since these are not available in the embedded programming language of Coq. These considerations led us to the following design choices:

- We compute Γ_i (step iv) incrementally, like Robertson *et al.* [26] (Appel and Haken could not do this [4], because computers in the early 70s did not have enough core memory to store a million bits…)
- We use 3 and 4-way decision diagrams [1,12] to store Θ_i and Γ_i, respectively; each level in the decision tree branches according to a color or symbol in the sequence or chromogram, respectively.
- We store the complements Ξ_i and Λ_i of Θ_i and Γ_i, respectively, rather than Θ_i and Γ_i. This way the computation starts with large sets that are rapidly pruned, rather than with small sets that expand quickly. In addition, the large initial sets Ξ_0 and Λ_0 have a very regular structure, so their decision trees require very little memory, as most of their subtrees can be shared.
- Ξ_0 is not the set of $\{1,2,3\}$ sequences whose length is the ring size, but the set of all such sequences whose bitwise sum is 0, and in which the order in which colors appear is a cyclic permutation of $\langle 1,2,3\rangle$, e.g., if the first color is 3, the first color different than 3 will be 1. The first restriction rules out sequences that can't possibly be induced by edge colorings; the second one is a free way to halve the size of the set, based on the observation that the "consistent" relation $\approx$ is insensitive to swapping colors 2 and 3.
- Likewise, Λ_0 is the set of all chromograms that are consistent with some $\theta \in \Xi_0$. This corresponds to excluding ill-bracketed words, and words for which the bit sum of the closing bracket parities does not match the parity of the ring size.
- To avoid rescanning the large $\approx$ relation in step v, Ξ_i actually stores the number of chromograms in Λ_i consistent with each θ it contains, which allows step v to be carried out in constant time. Thus, unlike the Robertson *et al.* program [26], our fix point computation runs in time linear in the size of the $\approx$ relation, traversing each $\theta \approx \gamma$ pair at most twice.
- We use dynamic programming to compute a highly compressed representation of Ξ_0 and Λ_0, but do not try to find ad hoc sharing afterwards; initial experiments with an ML prototype had showed that there was little to gain. We do however omit the last layer of Ξ_i, as the last color in a valid sequence is always the sum of the previous ones, and we also use a fixed compression scheme for the last two layers of Λ_i, using 8 fixed constants (since this is 4-way branching tree, this saves 75% of the memory!).

The computation iterates the three following steps:

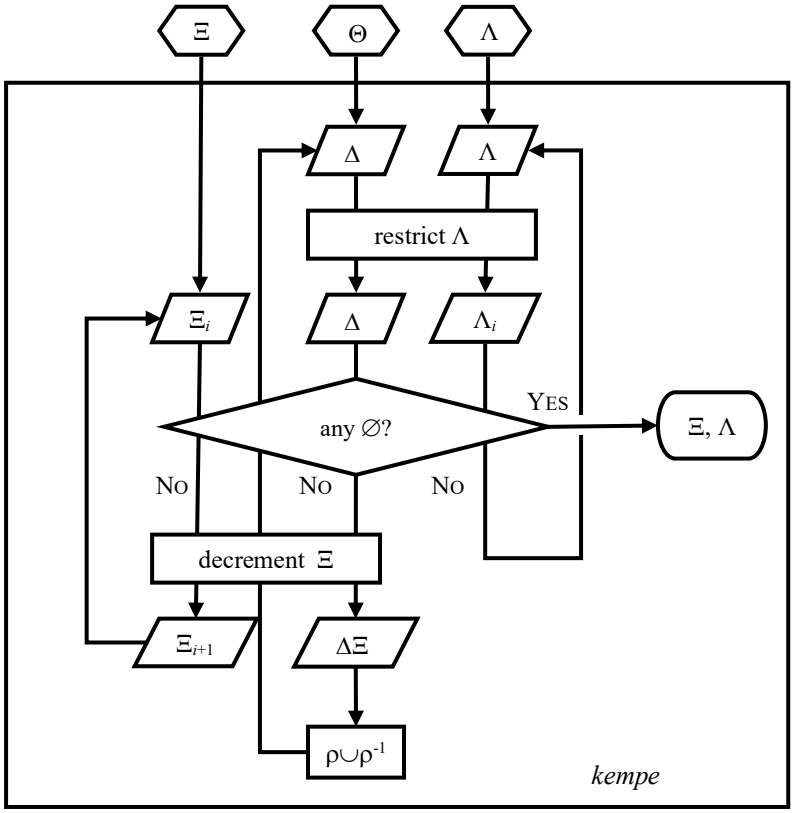
i. Given a set of color sequences $\Delta\Theta$ to be added to Θ_i, remove all the chromograms in Λ_i that are consistent with some $\theta \in \Delta\Theta$, simultaneously computing Λ_{i+1} and $\Delta\Gamma = \Lambda_{i+1} - \Lambda_i$, the set of chromograms to be added to Γ_i.

ii. For each θ in the domain of Ξ_i, decrement the value of $\Xi_i(\theta)$ by the number of chromograms in $\Delta\Gamma$ consistent with θ, simultaneously computing $\Xi_{i+1}(\theta)$ and $\Delta\Xi = \{\theta : \Xi_i(\theta) > 0 \text{ and } \Xi_{i+1}(\theta) = 0\}$, the set of color sequences whose count has just reached 0.

iii. Compute $\Delta\Theta = \{\rho^e\theta : \theta \in \Delta\Xi, e = \pm 1\}$, and feed it back to step i. Here ρ is the cyclic permutation that maps $\langle 1\ 2\ 3\rangle$ to $\langle 2\ 3\ 1\rangle$.

We start with $\Delta\Theta = \Theta_1$ and terminate as soon as one of Ξ_i, $\Delta\Gamma$, or Γ_i becomes empty. All the intermediate sets use the same MDD representation, so steps i-iii can all be implemented efficiently with parallel tree walks. Finally, the fix point loop is driven by a higher-order tree iterator that can efficiently execute an exponential number of steps.

While this is clearly overkill (the number of iterations never goes over 30), it does allow us to do a full correctness proof of the algorithm, sparing us the hassle of having to debug the execution of such a complex algorithm on large data structures in a development environment with absolutely no debugging support (not even print statements!). Furthermore, the full correctness proof is surprisingly straightforward; it is carried out mostly by stepping through all the cases of all the functions.



We also needed to prove that the formal, purely combinatorial definition of "suitable" actually implies something about the coloring of planar maps. In principle this implied formalizing all the theory that we exposed in this section: Tait's edge colorings, the topology of the chords, and the parsing of a chromogram into a chord set. Benjamin Werner, who worked with us on this part of the proof, found out that none of this was needed, except as insight into the solution. It was much easier to prove directly that if a remainder map has a suitable coloring, then it has a coloring that matches a border coloring of the configuration; the main lemma, which states the existence of a consistent chromogram, is proved by induction on the size of the remainder map. Indeed, this result can be proved directly from the Euler formula, whereas we needed to use the Jordan property *twice* in our informal discussion! When we finally decided to port his outline to our development on hypermaps, it only took us a day to complete the proof[10].

5.3 Configuration maps

Since the reducibility check is based solely on the edge-coloring of the configurations and their contracts, it would appear that it is not necessary to describe the exact

[10] We cheated by reusing the `euler_tree` lemma, which we had used to prove Jordan→ Euler.

geometry of the maps to prove the Four Color Theorem: the colorings can be computed from the *adjacency graphs* of the configuration maps, and in the unavoidability part of the proof these adjacency graphs are matched against the graphs of the explicitly enumerated second neighborhoods submaps of an arbitrary counter example; the latter are, by construction, geometrically correct. This is how Robertson *et al.* outline their proof [26], and indeed they use adjacency lists to describe their 633 configurations. We naively followed this line of reasoning in our initial investigations, which focused on the feasibility of the reducibility proof, since it was straightforward to compute the set of colorings of a graph. However, when we started working on the graph theory linking the two computational parts of the proof, we realized that this approach was wrong:

- Several small geometric side conditions are imposed on configurations and their contracts; while most of them are simply sanity checks, some are used in the actual proof, and therefore must be checked formally.
- Enumerating just the arity of faces in neighborhoods is already fairly complex; enumerating their full geometry would have been one step up.
- Most of all, the enumeration strategy proposed by Robertson *et al.* does not explicitly identify complete submaps, because it is limited to the second neighborhood, and the configurations needed to prove the Four Color Theorem do not lie entirely within the second neighborhood: only their kernel does. This is sufficient, if one can apply the "folklore theorem" that any kernel has a unique free ring completion; however, this theorem only applies if the adjacency graph of the completion is geometrically consistent.

Of course, the Robertson *et al.* data *is* geometrically consistent: it gives a planar embedding for each configuration, for which adjacency lists are oriented clockwise. The programs supplied by Robertson *et al.* also perform a combinatorial sanity check.

Nevertheless, we had to revise our approach, and explicitly define a hypermap for each configuration. We first did this directly from the adjacency data, using pairs of integers for the darts; this allowed us to keep the coloring and contract coloring functions, along with their correctness proofs. However, this approach also involved encoding geometric reasoning as obscure arithmetic identities, and didn't support very well doing induction on the geometry of the map, which was required for establishing the correctness of the geometric side condition checks.

A little analysis revealed that each of the 633 configuration submaps could be built by following a very simple, inside-out construction program, so we switched to that description. This change cut down the size of the configuration data by a factor of 5, but, more importantly, it provided a clean framework for the various checks we needed to perform on the maps. We could rephrase all the operations we needed to perform on a configuration map (computing the set of colorings, the contract, checking the radius, even compiling an occurrence check filter) as nonstandard interpretations of its program (the standard interpretation being the map construction). This approach both made the implementation efficient, and the correctness proofs straightforward, although sometimes lengthy: these proofs always consisted in establishing some simulation relation between the standard and nonstandard interpretations.

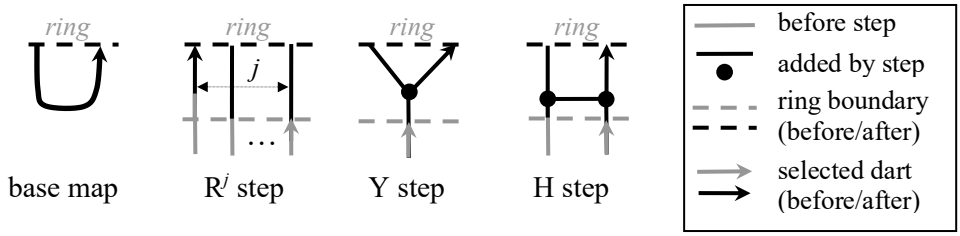
Although the main use of configuration maps is the reducibility argument, which calls for *disk* maps, we chose to construct *remainder* maps, for various technical reasons:

- The proof of the Birkhoff Lemma [10] requires a handful of explicitly constructed remainder maps (see §5.4 below).

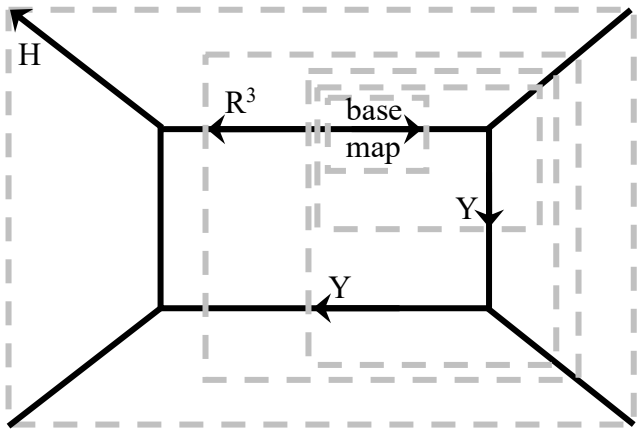
- It's easy to show that the construction yields a remainder map with a simple border n cycle; stripping this cycle trivially yields a disk map with the same colorings.
- The remainder map construction is more uniform because it starts with a non-empty map.
- Remainder maps are plain and quasicubic (cubic except on the n cycle) whereas disk maps are cubic and quasiplain (plain except on the e cycle), and the plain identities, $e(e(x)) = x$ and $n(f(x)) = e(x)$, are more useful than the cubic ones.

The construction actually yields a *pointed* remainder map, in which one dart x on the border n cycle has been selected. The construction starts from a single edge and applies a sequence of steps to progressively build the map. For configuration maps, it turns out we only need three types of steps:

- R^j (rotate) steps that only change the selected ring dart x to $n^{-j}(x)$ for some j.
- Y steps that insert a node between x and the border ring.
- H steps that insert a linked pair of nodes between x and $n^{-1}(x)$, and the border ring.



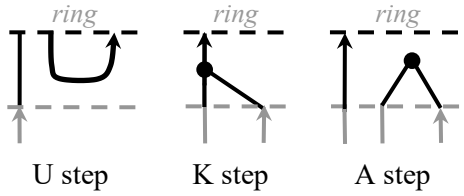
The first construction step is always a Y step. Construction sequences are applied right to left, so the construction sequences for configurations always end with a Y (they also start with an H). For example, the program $\langle H\ R^3\ Y\ Y\rangle$ constructs a map consisting of a single square surrounded by a ring of four faces:



The data for a configuration also includes a contract, that is, a set of edges that should be erased to get, by induction, a suitable coloring of the remainder. As these edges must not contain border ring darts, they must have been disconnected from the border ring and connected to an internal node by a Y or H step. We therefore denote contract edges by the left-to-right index of this connection, counting that each Y makes one such connection, and each H makes three (because the initial step is a Y, the feet of the H belong to different edges). These indices appear before the H that starts the construction sequences (the other indices in the sequence are R steps). This representation allows us to easily find contract edges while scanning the program left to right.

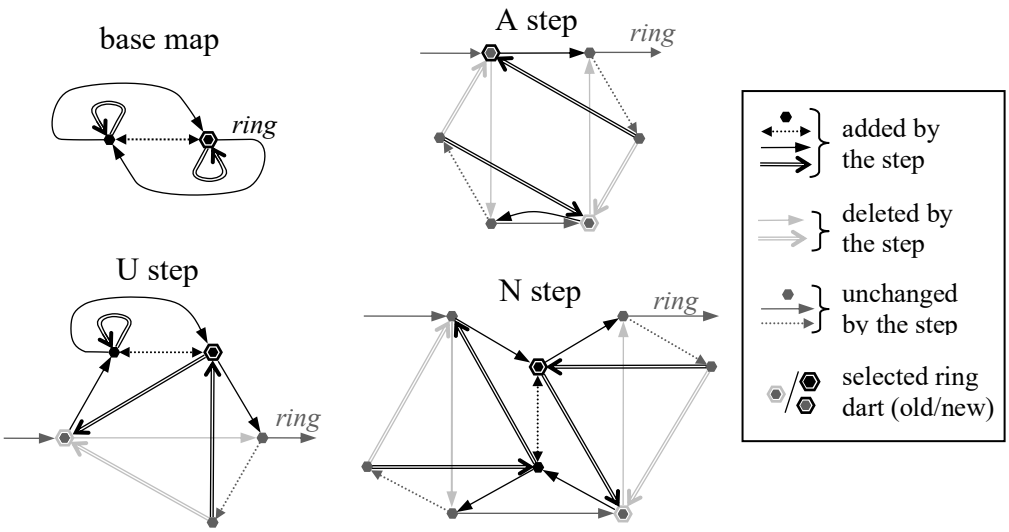
Since the correctness proof of the function that computes configuration border colorings is rather long, we decided not to try to combine it with the definition of contracts. Instead, we added three new construction steps that allow us to construct a map that had the same set of colorings as the contract map (or, more precisely, the same "contract coloring" as the configuration map). This allows us to produce the set of contract colorings by compiling the configuration program into a contract program, then running that program with the "generate colorings" interpreter. The new steps are:

- the U step, which inserts a single loop between x and $n(x)$
- the K step, which adds an inverted Y above x and $n^{-1}(x)$
- the A step, which connects x and $n^{-1}(x)$ with a degree 2 node (the resulting map is not quasicubic; in fact, it might not even be plain – see below)

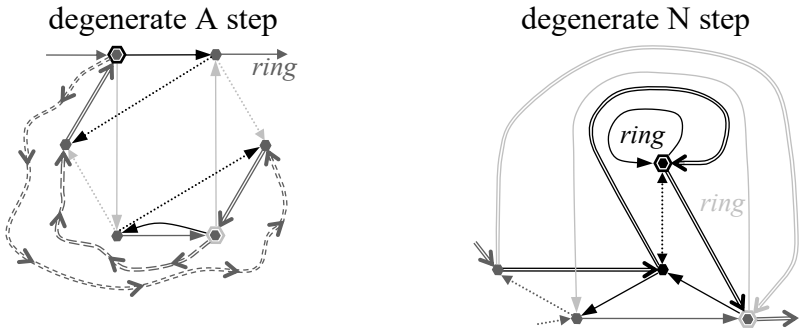


We also use the U to construct some of the remainder maps used in the proof of the Birkhoff lemma [10].

These new steps are more primitive than the H and Y steps; indeed, we can *define* the H, Y, and K steps in terms of U and a rotation variant $N = R \circ K \circ R^{-1}$ of K, as we then have $Y = N \circ U$, $H = N \circ Y$, and $K = R^{-1} \circ N \circ R$. Thus, we only need to give precise hypermap constructions for the base map and the U, N, and A steps:

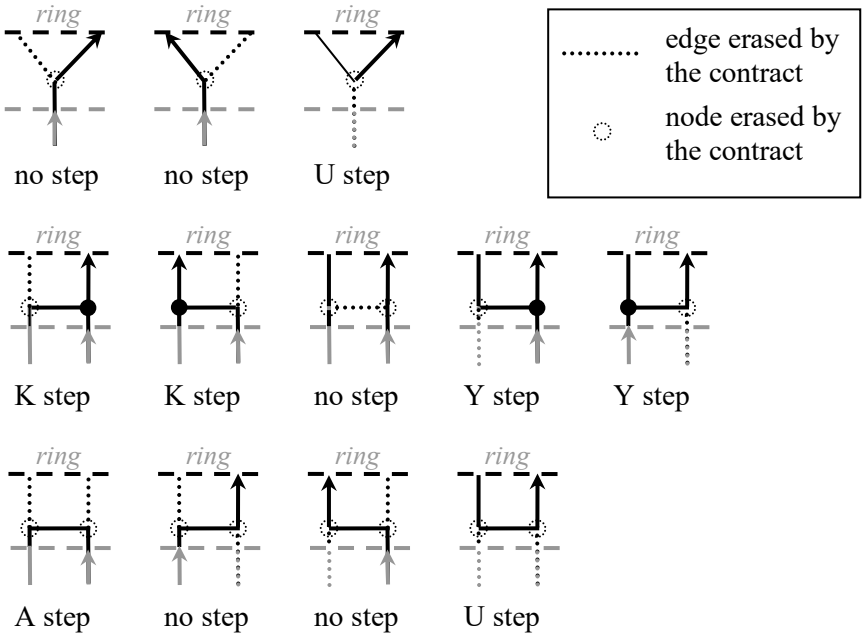


The A and N steps have degenerate cases in which these definitions are incorrect. For the N step, this occurs if the ring size is less than three: if x is the selected dart, we have $n^{-2}(x) = x$, but the definition above gives different values for $n'(x)$ and $n'(n^{-2}(x))$.
The degenerate case for the A step occurs when x and $n^{-2}(x)$ are already on the same f cycle: in that case the definition above splits the cycle and disconnects the hypermap (this is similar to the split case of the Walkup transformation). Although these degenerate cases do not occur with our data, this is not easy to check for the A step, and at any rate is easier to provide consistent definitions for these cases than to deal with partial constructions throughout the proof, so we use the following:

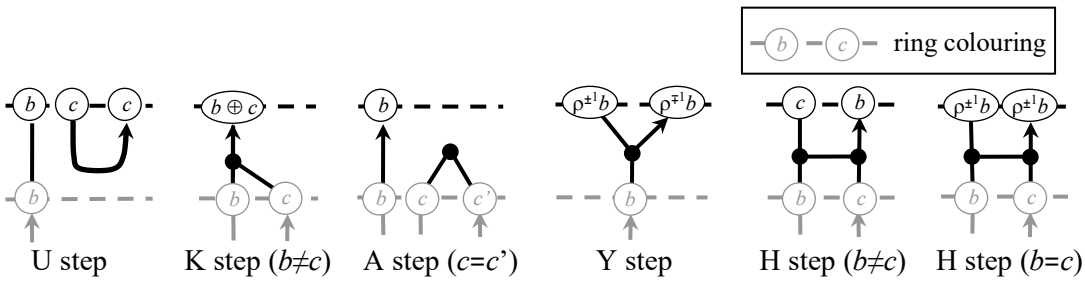


Note that both these definitions are still valid in the completely degenerate case where the ring size is 1.

The contract compilation is defined in the figure below, which shows the effect of erasing one or two edges from a Y or H step. The correctness proof of the contract compilation would seem straightforward from these diagrams, but it is the longest proof (close to 700 lines of script) in the entire development, with the second largest compilation time (excluding the reducibility and unavoidability computation lemmas).



The correctness proof of the edge coloring procedure defined in the figure below (where ρ is the permutation defined in §5.2) is comparatively easier, because we can use the correctness lemmas for the U and K steps to show the correctness of the Y and H steps, as the Y and H steps can be defined in terms of U, K, and R steps.

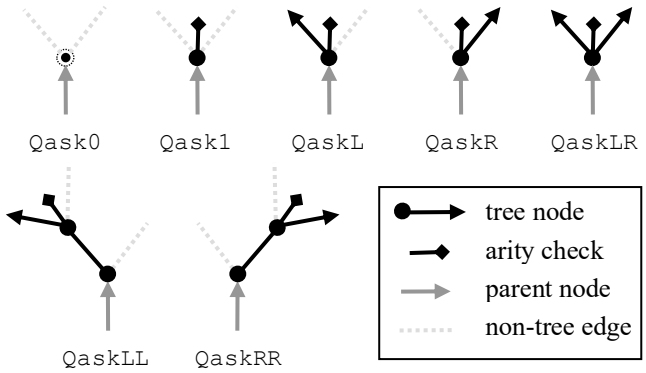


Observe that while some of the coloring rules have side conditions, only the A and K step can fail; hence the coloring of a non-contracted configuration never backtracks.

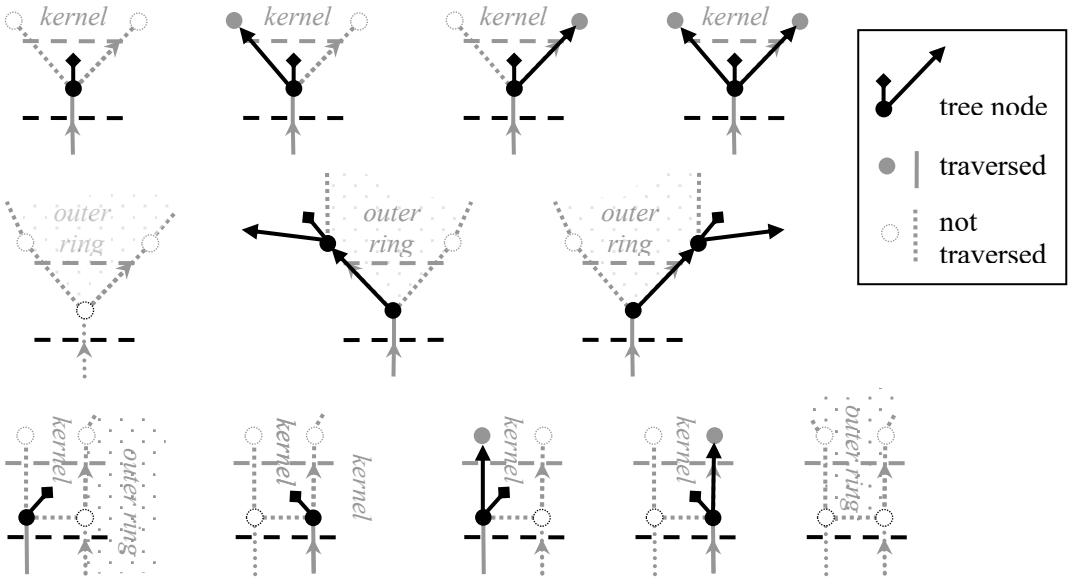
The unavoidability part of the Four Color Theorem proof requires yet another interpretation of the map construction programs: we need a filter that will efficiently test for the occurrence of a configuration in the data structure used to represent a second neighborhood pattern, which, following Robertson *et al.,* we shall call a *part* [26]. Now Robertson *et al.* computed this test by somewhat arbitrarily assigning integers to faces of the configuration and part, precomputing the geometrical relations (in their case, the clockwise triangles) for both maps, and then using yet another array to store the partial homomorphism in a traversal of a list of triangles of the configuration [27]. As we have pointed out already, the Coq language does not have arrays, and the idea of emulating them in order to implement such an indirect algorithm, and proving the correctness of the combination, was not appealing. It seemed that a more geometrical approach should exist, and we did indeed find one, by combining the insights provided by our analysis of hypermaps and of the incremental construction of the configuration maps.

As we shall see in the next section, we strengthened one of the auxiliary results of Robertson *et al* [27]. and showed that if it was possible to perform a geometrically connected traversal of the faces of the *kernel* of the configuration in the part, while generating the same sequence of arities, then the geometrical properties of configurations and counter example maps directly implied the existence of a one-to-one embedding of the disk map of the configuration into any counter example map fitting the part. It follows that there is no need to construct the isomorphism, hence, no need for arrays.

All we needed was a data structure to represent a map traversal, and we found that the most obvious one worked. We ensure that our traversal is connected by strictly following kernel edges of the map. Each node in the traversal must be cubic, since it is reached by a kernel edge, so there are two other edges leaving that node, and a face across those two edges, whose arity might need to be checked. The traversal can end with this check, or continue with either or both of the two edges. It is also possible that the face across is a ring face; in that case the arity check must be skipped by the traversal as there may not be a corresponding face in a matching part. This will certainly happen once for the many configurations whose kernel is not two-connected, and in fact this is the only case in which this happens. In those instances, the traversal should continue either left or right, check the face there (it's always a kernel face) and then continue in the same direction (that is, the traversal does either two lefts or two rights). Hence, the structure we need is a 7-case variant of the binary tree, shown below.



In turns out that it is always possible to cover exactly the configuration map kernel, using two such trees, while starting from the two faces incident to the initial edge of the construction. Thus, we can generate an optimal test (called a `quiz` in the Coq scripts) for all 633 configurations. We compile this pair of trees outside-in, that is, by interpreting the construction program in *reverse*, left-to-right, using the rules in the figure below. Conveniently, this is also the right direction for computing the face arities, which are stored in the tree nodes, and for keeping track of which ring darts belong to the kernel in the full configuration map, and which ones belong to the border ring.



Note that the rules above are not quite complete: we do not handle the case where there are non-empty trees for both arms of an H, which would have required a `QaskLRR` tree node (the other apparently missing cases can be ruled out using the geometric properties of configurations). It turned out that only 4 of the 633 configurations required that case, so it was easier to do a trivial modification of those 4 construction programs than to prove the correctness of an additional complex case.

5.4 Embedding configurations

The main result of the graph theory part of the Four Color Theorem proof is that no dart of a minimal counter example can fit the quiz of a (combinatorially) reducible configuration. The few weeks it took us to prove this represented only a fraction of the effort involved; much more time was spent developing and experimenting with the

basic concepts – sequences, paths, hypermaps – to see which formalization would allow us to complete this part of the proof. This attention to detail paid off in the end: it turned out that half of the final part of the proof could be skipped.

Specifically, we were able to eliminate *cartwheels*, an intermediate combinatorial structure used to represent second neighborhoods. Robertson *et al.* use cartwheels as follows [26, 27]:

- They recall the Birkhoff Theorem [10], which states that second neighborhoods in a minimal counter example are isomorphic to cartwheels.
- They do the unavoidability enumeration on cartwheels, defining both part matching and discharging on cartwheels.
- Although they use an array-based version of our quizzes to check for configurations occurrence in parts, they only do so as a heuristic. A positive check is always backed by the explicit construction of an injective simplicial map from the configuration kernel to the "skeleton" of the part, as well as an ad hoc check that the configuration is "well-positioned". They state without proof that these checks imply the existence of a morphism from the configuration to the map.

Since we were bent on doing a fully formal proof, using cartwheels would not give us the benefits of reusing well-known results. On the contrary, they would just introduce another layer of definitions and of correspondence lemmas, which were not really needed. Since it is easy to navigate precisely in hypermaps (just use the right sequence of *e*, *n*, and *f*), we could define discharging and part fitting directly on the full counter example map just as easily as on a cartwheel (in fact we derive discharging from part fitting). More importantly, we could do completely without the Birkhoff theorem, by showing directly that the mapping derived from a successful quiz check was automatically an embedding. This was a boon, because it relieved us from having to prove in every case the injectivity of all the intermediate mappings – from configuration to part, and part to map – that composed this mapping: we only needed to show that these were homomorphisms.

To summarize, we used the following proof outline:

1. We prove the Birkhoff Lemma [10] (every minimal counter example map is internally five connected).
2. We deduce from this that any face of such a map is surrounded by a simple ring (called a *spoke ring*) of at least five faces (we say the map is *pentagonal*).
3. Given a successful quiz, we construct a *preembedding* ϕ from the kernel darts of the configuration map to the counter example map, that is, a mapping that preserves all *f* arrows and arities, and preserves enough *e* arrows to connect its domain (i.e., the kernel is connected by the relation whose graph is the union of the *f* arrows and the set of *e* arrows $x \to e(x)$ such that $\phi(e(x)) = e(\phi(x))$.
4. We show that, provided the configuration map satisfies two simple geometric conditions, ϕ can be extended into an embedding $\tilde{\phi}$ of the entire configuration map, injective except on the border *n* ring.
5. Following Robertson *et al.* [26], we don't try to replace directly the image under $\tilde{\phi}$ of the contract map, as we would have no simple way of proving that the resulting map is bridgeless – the contract map need not be chordless, and its border ring may not even be simple. Rather, we apply directly the image under $\tilde{\phi}$ of the configuration contract to the counter example map, and show that coloring this map induces via $\tilde{\phi}^{-1}$ a coloring of the contract map. We show that the geometric properties of the

contract are carried over by $\tilde{\phi}$, and that they imply that applying the contract to the counter example map yields a bridgeless map.

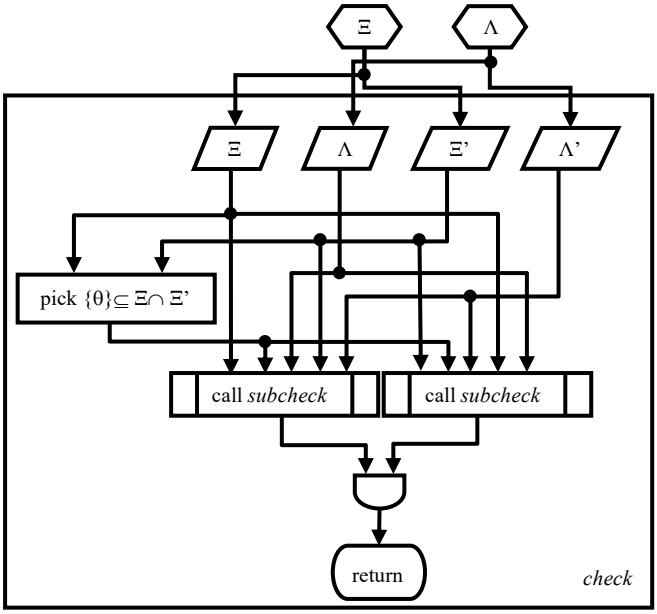
6. We conclude as indicated in subsection 5.2.

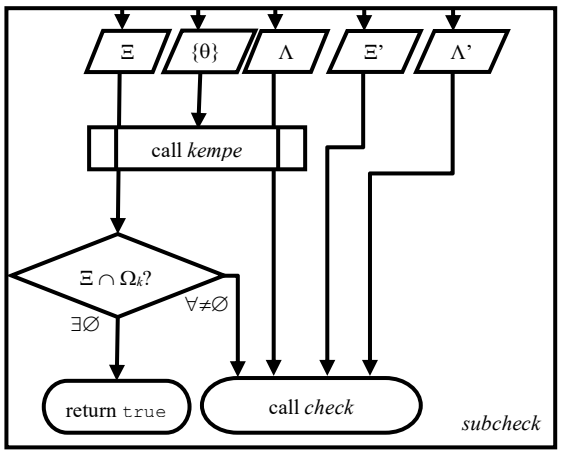
For each step except the last, we ended up inventing our own proofs.

For Step 1, we use a variant of the reflection method used for reducibility, reusing much of the code from that part of the proof. The formal statement for Step 1 is "there are no simple *m*-nontrivial rings of size $k \le 5$", where m=1 if k=5, and m=0 if $k < 5$, and a ring is *m*-nontrivial if both its inside and outside have more than *m* faces.

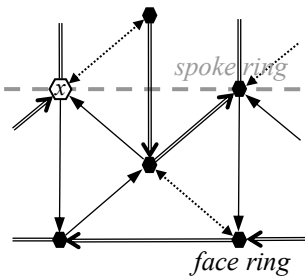
Let Ω_k be a list of configuration maps with ring size k and at most m kernel faces. Suppose r be such an m-nontrivial ring of size k in a minimal plain counter example map g, and consider the set Ξ (resp. Ξ') of border colorings of the remainder map h (resp. h') of the inside (resp. outside) of r. Since g is plain and not four colorable, Ξ is disjoint from the set Θ of colorings suitable for h, hence disjoint from the safe approximation of Θ computed from Ξ' as in §5.2. Thus, by symmetry, we must both have $\Xi \subseteq kempe(\Xi_0,\Xi',\Lambda_0)$ and $\Xi' \subseteq kempe(\Xi_0,\Xi,\Lambda_0)$. We check by enumeration that for any such pair Ξ, Ξ' there is some $h'' \in \Omega_k$ whose set of border colorings Ξ" is disjoint from either Ξ or Ξ'. This contradicts the minimality of g, since if, for example, Ξ and Ξ" are disjoint, then the map obtained by replacing h' with h" in g is strictly smaller but not four colorable.

Using the monotony of *kempe*, the enumeration of the Ξ, Ξ' pairs and the Ω_k checks are carried out simultaneously by the two mutually recursive procedures below (the flowchart shows only the successful path, where $\Xi \cap \Xi'$ is never empty in *check*).





While the lemmas proved in Step 2 are not original (they are a subset of the consequences of the Birkhoff Lemma), their proofs are somewhat different from traditional mathematical argument, because they rely on the (abundant) computing power of Coq rather than its (non-existent) geometrical intuition. For example, we establish exact formulas for navigating about the first neighborhood of a face: e.g., x is a dart on the spoke ring if and only if $n(x)$ is on the central face cycle and the next dart on the ring is $f(f(e(x)))$. We then use these formulas to replace visual reasoning on figures with equational reasoning. To prove Step 2 we exhibit an explicit coloring function and use brute force enumeration to prove that it's a valid coloring; this way we don't need to prove that a chordless 4-face ring must be 0-nontrivial, as in steps d) and e) in Section 3.



For step 3, we define ϕ as follows: let $x_1 \ldots x_n$ be the sequence of darts traversed by running the quiz on the configuration map (marked with black diamonds ♦ in the figures). Since this sequence contains exactly one dart in each face cycle in the kernel of the configuration map, for any dart x in this kernel we can find a unique i such that $x = f^k(x_i)$ for some k. We then set $\phi(x) = f^k(u_i)$, where $u_1 \ldots u_n$ is the sequence of darts traversed by running the quiz on the counter example map. Since both runs succeed, x_i and u_i have the same arities, so ϕ trivially preserves arities and f arrows. Since the two `question` trees composing the quiz span the configuration map kernel, we can then establish that ϕ is a preembedding by showing that it preserves one e arrow for each branch in these trees. This follows from the equational characterization of the "left" and "right" steps that were presented graphically in 5.3:

- a "left" step takes us from x to $n(e(n(x))) = e(f^{-2}(x))$
- a "right" step takes us from x to $n(e(x))) = f(e(f(x)))$

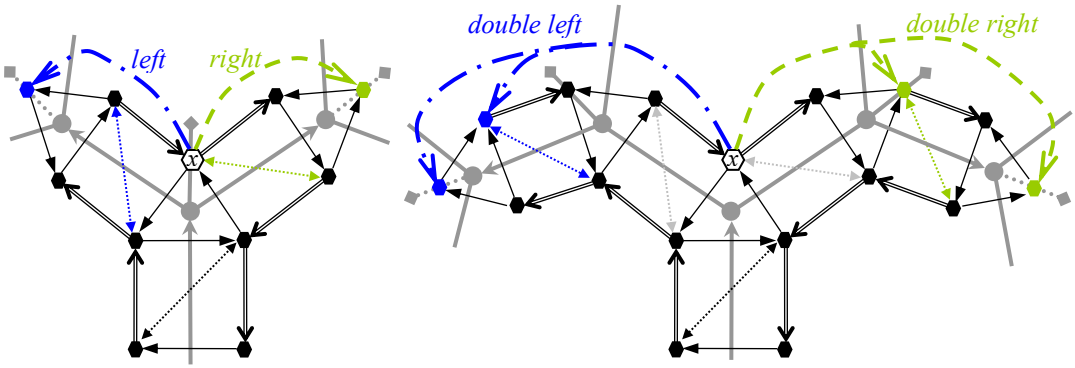
However, we can't use these equations directly, because the traversal may leave the kernel during double left and double right steps. We get around this issue by slightly adjusting our definition, setting the traversed x_i one f step before the ♦-marked dart for double-left steps (see the figure below). This ensures that the following property holds

- $\phi(e(x_i)) = e(\phi(x_i))$ for every x_i in $x_1 \ldots x_n$.

This follows from a property of the `quiz` tree traversal

- $\phi(e(n^{-1}(x)) = e(\phi(n^{-1}(x))$ for every x at which the recursive `quiz` tree traversal is called (because of the double steps, x is not necessarily one of the x_i).

This holds at the tree roots because the traversals start from an edge cycle in both the configuration and counter example maps, and inductively because of the geometric identities.



Since ϕ preserves f arrows, the first of these equations implies that ϕ preserves at least one e arrow connecting the face cycle of x to the face reached by a simple left or right step. The second equation is used for the double-left and double-right steps; it implies the existence of another e arrow connecting the face reached by the double step to the lower part of the tree.

Step 4 is new to our proof, although it was inspired by the justification for the configuration search heuristic of Robertson *et al* [27]. The two additional geometrical constraints on configuration maps are:

a) The kernel must have radius 2, that is, there must be a face in the kernel whose second neighborhood contains the entire kernel.
b) The arity of the ring faces must be between 3 and 6, inclusive.

Condition a) is fairly natural, since we are trying to embed the kernel of the configuration map inside a second neighborhood of a counter example map; condition

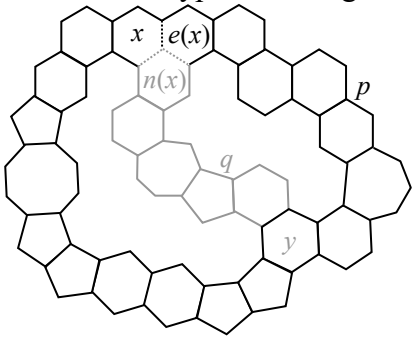
a) was mentioned by Robertson *et al.* Condition b) is analogous to Robertson *et al.*'s "well-positioned" condition, but is simpler in that it does not depend on how the configuration embeds in the second neighborhood. We have an informal argument that condition b) should hold for all configurations useful for the Robertson *et al.* proof technique, but our proof does not depend on this argument, since we explicitly check for condition b) during the quiz compilation; we check condition a) separately.

The conclusion of step 4 rests on three geometrical lemmas:

i. ϕ preserves all e arrows in the kernel.
ii. ϕ maps kernel e arrows faithfully.
iii. The border ring of the configuration is chordless.

The proof of each of these uses induction. The proof of lemma i. depends only on the connectedness of the domain of ϕ, and the preservation of f arrows; it doesn't use conditions a) or b), or arities. It proceeds as follows:

- Consider a kernel dart x such that $e(x)$ is also in the kernel, but $\phi(e(x)) \neq e(\phi(x))$.
- By the connectivity property of ϕ, there must be some simple path p of faces in the kernel that goes from $e(x)$ to x, and is preserved by ϕ; i.e., p is a ring in the configuration map kernel, but $\phi(p)$ is only a path in the counter example map.
- We use induction on the size of the disk map delimited by p, i.e., on the number of nodes (n cycles) in the interior of p. By exchanging x and $e(x)$, we may assume that this disk map lies inside the kernel.
- Consider the arrows $n(x) \leftrightarrow e(n(x)) = f^{-1}(x)$ and $f(e(x)) \leftrightarrow e(f(e(x)) = f^{-1}(n(x))$. If they are both preserved by ϕ, then we get $\phi(e(x)) = e(\phi(x))$ by equational reasoning, using the fact that the counter example map is cubic. This contradicts our assumption.
- Hence, one of these e arrows is not preserved; note that these correspond to the edges between the face containing $n(x)$ and those containing x and $e(x)$, respectively.
- The face containing $n(x)$ is on or inside p, so $n(x)$ is in the kernel, and there must be a simple path q, not meeting p, from some face $y \in p$ to $n(x)$ (y may be x or $e(x)$). Dividing p at y and combining one of the pieces with q gives us a ring with a smaller disk that does not contain the node between x, $n(x)$ and $e(x)$, so we can apply the induction hypothesis to get a contradiction.

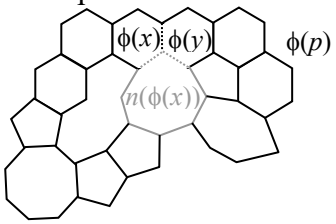


The proof of lemma ii. follows a similar argument, where the roles of the configuration and counter example maps have been swapped.

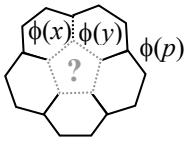
- We want to prove that if $\phi(x) = e(\phi(y))$ for some kernel darts x and y, then $x = e(y)$, so assume some x and y for which this does not hold.
- Condition a) implies that there a simple path p in the kernel of the configuration map going from y to x, of length at most 5 (at worst, we go from x to the central

face and then to y). Because of i. and the assumptions on x and y, $\phi(p)$ is a ring in the counterexample map, while p is only a path in the configuration map.

- Exchanging x and y and reversing p if necessary, we can assume that the interior of p is no larger than its exterior.
- If $\phi(p)$ has an empty interior then we can use a simplified variant of the induction used for lemma i. to derive a contradiction ($n(\phi(x))$ must be in $\phi(p)$). We will reuse this fact to prove lemma iii. below.



- By the above, $\phi(p)$ is 0-nontrivial, so by the Birkhoff Lemma it must have length 5 and its interior must consist of a single face.



- Thus ϕ maps p to a spoke ring; since ϕ preserves the formula for the next dart in a spoke ring, p must be a spoke ring arc in the configuration map, wrapping around a single face of the configuration map. That face cannot be in the kernel, for then it would have to be a pentagon, since it is mapped to the central pentagon and ϕ preserves arity on the kernel, and this would imply $x = e(y)$. Hence it is a ring face, adjacent to the five different faces in p. Since it must also be adjacent to the faces respectively preceding and following it on the border ring, it must be at least a heptagon; but this violates condition b).

The proof of lemma iii. uses yet another variant of the same argument:

- Any chord of the border ring of the configuration map splits it in two rings with disconnected interiors. Since the union of these interiors is the kernel, which is connected, one of the subrings must have an empty interior.
- As in the proof of lemma ii. above, by induction on the length of the subring with an empty interior, there must be a subring of length 3.
- The middle face of that subring is only adjacent to two other faces in the subring – this violates condition b).

The combination of lemmas i–iii. implies that ϕ faithfully injects the kernel of the configuration map into the counter example map. Since the configuration map is plain, and cubic except on the border n cycle, we can define

$$\tilde{\phi}(e^i(n^j(x))) = e^i(n^j(\phi(x)))$$

to get the required embedding: it follows from iii. that each border ring face is adjacent to at least one kernel face, hence the equation above defines $\tilde{\phi}$ totally.

Finally, for step 5 we follow closely Robertson *et al.*, using the following conditions to ensure that the application of a contract yields a smaller bridgeless map:

a) The contract is *sparse*: no two darts of its e-closure belong to the same n cycle, or to the border n cycle.
b) The contract has between 1 and 4 darts.

c) If the contract has 4 darts, then it has a triad: there is a kernel dart of the configuration map that is adjacent to at least 3 darts, belonging to different faces in the *e*-closure of the contract, but not to all such darts.

Our condition c) is slightly more restrictive but easier to check than the corresponding condition in [26]. We show that conditions a), b), and c) are preserved by $\tilde{\phi}$, with the non-border and kernel requirement dropped in a) and c), and that the combination of a), b), and c) implies that no simple ring *r* of a minimal counter example can consist almost entirely of contract edges, i.e., be such that all but one dart of *r* belong to the *e*-closure of the contract:

- By induction on the size of *r*, condition a) implies that *r* cannot have an empty interior; the proof is similar to that of lemma iii. in step 4.
- By symmetry, *r* is thus 0-nontrivial; hence by the Birkhoff Lemma and condition b), *r* must be the spoke ring of a pentagon, and contain all the darts of the *e*-closure of the contract, which must be of length 4.
- Condition c) states that there is a dart adjacent to three different darts in *r*, yet is not in the inner pentagon cycle (it would then be adjacent to all of *r*). This violates a corollary of the Birkhoff Lemma.

We conclude by showing, by induction on the contract size, that the absence of such almost-contract rings implies that erasing all the contract edges yields a smaller bridgeless map, whose coloring induces a contract-coloring of the counter example map, and thus of the contract map.

5.5 Enumerating parts

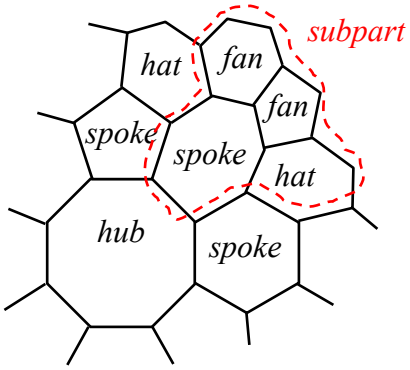
Recall that the unavoidability proof consists in proving that in a minimal counter example the second neighborhood of any face whose "discharged" arity is less than 6 contains a copy of the kernel of one of the 633 reducible configurations. This is done by going through a symbolic enumeration of all possible shapes of second neighborhoods, checking for each shape that either

a) The central face has arity at least 6, after "averaging" with the discharge rules.
b) The `quiz` for one of the 633 configurations runs successfully at one of the darts of the neighborhood.

Both of these checks depend only on the arity of the neighborhood faces, so a "shape" is just a mapping from these faces to integer intervals. We call these mappings *parts,* following Robertson *et al*. Using their terminology [26], we distinguish four kinds of faces in the domain of a part:

- The central face is called the *hub*; it always has a fixed arity.
- Faces in the first neighborhood of the hub are called *spokes*.
- Faces in the second neighborhood that are adjacent to two (consecutive) spokes are called *hats*.
- Other second neighborhood faces, which are adjacent to only one spoke, are called *fans*.

We represent a part as a counter clockwise sequence of *subparts*, where each subpart is a record containing the arity intervals for a spoke, the hat clockwise from it, and the counter clockwise sequence of fans that follows that hat. The length of the subpart sequence is the arity of the hub.



The intervals in a part take only a very limited range of values:

- The lower bound is at least 5, since a counter example map is pentagonal.
- The upper bound is either less than 9 or infinite, because the discharge rules don't discriminate between faces with more than 8 sides, and configuration maps have at most one face with more than 8 sides, which it is only matched with the hub. For the same reason, the lower bound is less than or equal to 9.

Hence there are only 15 possible intervals, and the interval $[5,+\infty)$ means "no constraint". We always assign the $[5,+\infty)$ interval to fan faces that are not known to be distinct, for example when the exact arity of the corresponding spoke is not known. Indeed, we have four kinds of subpart records, according to whether there are 0, 1, 2, or 3 known distinct fans; in the latter cases the arity of the spoke must be 6, 7, and 8, respectively.

We check that a part "fits" at a particular dart x of a counter example map by using the hypermap functions to navigate around the part and its subparts, verifying each nontrivial arity constraint. Crucially, this does *not* involve checking for a one-to-one correspondence between the faces listed in the part and the face cycles in the second neighborhood of x (by subsection 5.4, this is not needed). Because the definition of "fits" is purely equational, it can be manipulated efficiently by our Coq scripts.

Since "fits" relates parts to the counter example map, we can directly use parts to specify how arities are discharged. We can turn the 32 rules of Robertson *et al*. into a list $p_1,..,p_n$ of parts (n=71), and then define the discharged arity $\delta(r)$ of an f cycle r as

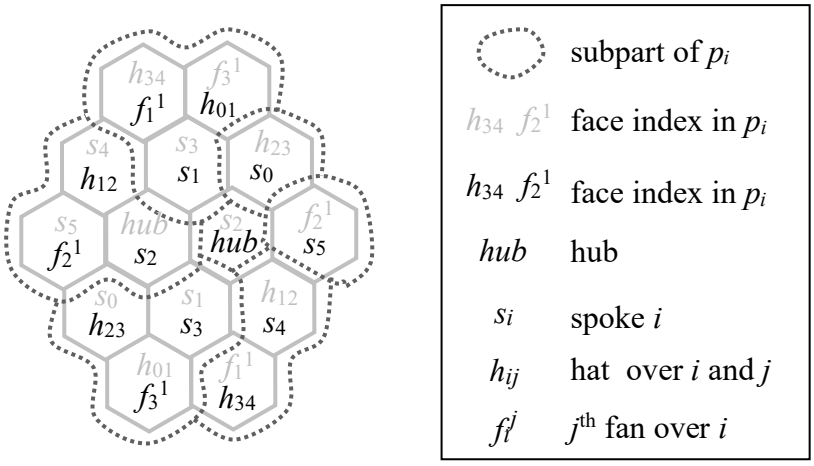
$$\delta(r) = \#r + {}^1/_{10}\sum_{x\in r} \#\{i : p_i \text{ fits } f^{-2}(x)\} - {}^1/_{10}\sum_{x\in r} \#\{i : p_i \text{ fits } f^{-2}(e(x))\}$$

Since e is an involution, we have, from the Euler formula,

$$\sum_r \delta(r) = \sum_r \#r = \#d = 6\#f - 12 < 6\#\mathrm{f} = 6\#\{r : r \text{ is an } f \text{ cycle}\}$$

Hence $\delta(r) < 6$ for some r, and this holds for *any* sequence p_i. In comparison, the definition used by Robertson *et al.* [26] depended on the Birkhoff theorem and on additional assumptions on the discharge rules.

Given a part p that fits a dart x, we can efficiently check whether some part p_i fits some dart $f^k(x)$ in the f cycle r of x by doing a member wise comparison of the intervals in p_i and p rotated left k times. Thus we can get a lower bound on the first sum in the definition of $\delta(r)$. To compute a lower bound on $\delta(r)$ we also need to find an upper bound on the second sum. As illustrated in the figure below, we use pattern matching to compute a part $\tilde{p_i}$ such that p_i fits $f^{-2}(e(x))$ only if $\tilde{p_i}$ fits $f^{-2}(x)$. (This is the reason for the f^{-2} in the definition of $\delta(r)$.) We can therefore get the desired upper bound by comparing $\tilde{p_i}$ with all rotations of p_i.

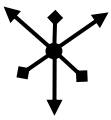


The above gives us an efficient way of checking for condition a). To efficiently run a quiz, i.e., to check condition b), we need to be able to navigate efficiently around a part. This is an issue because the Coq programming model does not include random-access arrays. However, running a quiz only requires *local* displacements (left and right steps), so we can use Huet's *zipper* data structure [21] to solve our problem. The data structure we use to search and check for reducible configurations is a *zpart* (for *zipped part*). A `zpart` is a representation of a pair of a part and of a dart position in that part, optimized for fast access to the arity intervals of nearby darts. It consists of

- the (counter clockwise) sequence of subparts that starts with the subpart s containing the dart position
- the *reversed* sequence of subparts before s
- a subpart location pinpointing the dart position in s

This representation makes it easy to move one step up or down the sequence of subparts in constant time, keeping the initial part p and its reverse in global constants to handle steps that wrap around the beginning or end of p (to minimize the occurrence of such cases, the total size of the forward and reverse sequences of a zpart is always *twice* that of p).

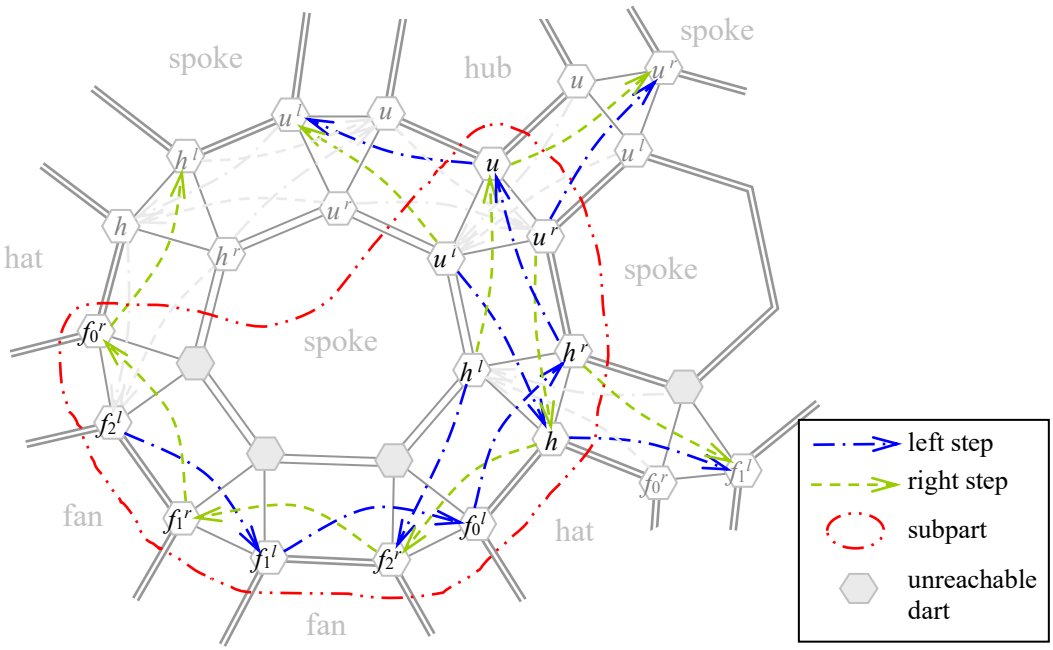
Because the construction of all our configuration maps always starts with a Y step, all quizzes start with a three-way branch from the (central) node created by the Y step:



Thus quizzes can be decomposed into the triplet of arities of the faces incident to the central node, and a triplet of `question`s rooted at the edges incident to the central node. We use a decision diagram structure to sort all the `question` triplets according their respective arity triplet, for all rotations and reflections of the central node (omitting reflections for configurations that have an axial symmetry). This speeds up considerably the configuration search:

- While traversing the part, we only need to consider *complete* nodes, such that all the three faces incident to the node are mapped to an interval of arities by the part. There are at most 6, and usually less than 3 complete nodes in each subpart.
- We only need to look up the triplet of the top values of these intervals in the decision diagram to get the list of `question` triples that should be tested. This list is usually quite short (it can contain up to 239 triplets, though).

In addition, a little analysis reveals that these `question`s only traverse a very limited subset of the dart positions of the part: only 14 positions are reachable in each subpart, so we can use a 15-element enumeration to represent subpart locations in a `zpart` (reserving one value for out-of-bounds locations). Coding and verifying the correctness of the configuration occurrence check was therefore a straightforward[11] exercise in brute-force enumeration that only took a few days.



We use the clever branch-and-bound strategy devised by Robertson *et al*. [26,27] to drive the enumeration of parts. They use explicit scripts to guide the initial decomposition of the part search space, then turn to either

i. blind enumeration to check that if we fix the arity of every face that is mapped to a *finite* interval by the part, then a reducible configuration occurs,
ii. automated branch and bound to check that $10\delta(r) - 60 \geq 0$.
iii. explicit matching to check that this branch is a subcase of the rotation/reflection of a previously explored branch.

Check ii. uses an explicit hint to break the bound check into checks that a well-chosen multiset of the

$$T[j] = \#\{i : \tilde{p_i} \text{ fits } f^{j-3}(x)\} - \#\{i : p_i \text{ fits } f^{j-3}(x)\}$$

and

$$T[j_1,j_2] = T[j_1] + T[j_2]$$

obey specific bounds that sum up to less than $20(m - 6) + 2$, where m is the arity of the hub x. The branching strategy for these searches consists in enumerating all the sub lists of $\tilde{p_i}$ that have a non-empty intersection with the appropriate rotation of the part p, and check that this intersection is included in enough p_i to respect the bound. If this fails, then check i. is performed on the intersection (in fact, most of the configuration searches occur this way).

[11] Except for a devious trick that allows double steps to wrap around a spoke whose arity interval is [5,6] – see file `redpart.v`.

Robertson *et al*. provide a single C program[27] that both interprets the explicitly supplied scripts and performs the automated checks i-iii. We used a hybrid strategy to implement this in Coq

- We used reflection, as we had done for the reducibility and Birkhoff lemmas, to perform the automated checks: we wrote (rather short) recursive Coq programs that performed the checks, and proved their partial correctness.
- We used a so-called *shallow embedding* for the explicit scripts: we added five Coq tactics that allowed us to directly interpret the explicit scripts[12] as Coq proof scripts.

We chose to use a shallow embedding mainly because it seemed silly to duplicate part of the proof search engine of Coq in a reflection program. However, this decision had another much more important, advantage: we could use Coq's interactive facilities to step through the scripts. This turned out to be crucial, because, although we had formally proved their correctness, most of our reflected functions needed to be debugged! The reason for this apparent contradiction was that we only formally verified *partial* correctness: that if a reflected check returned `true`, then the condition it checked for actually held. By that measure, a check that returned `false`, would be correct, albeit useless. As it turned out, a handful of trivial programming errors (e.g., off-by-one arithmetic, misspelled constants, and even clerical errors in entering the discharge rules) had crept in the development, and needed to be weeded out. As we pointed out earlier, debugging with a completely pure language is awkward, so the ability to step through the script was vital. Finally, the shallow embedding allowed us to use the proof assistant to develop our own scripts for parts of length 5 and 6. In the Robertson *et al.* proof these cases are dispatched with manual proofs using graphical arguments that would have been difficult to carry out in a fully formal system, so we decided instead to reuse the scripting infrastructure to come up with our own scripts. Although the script for size 6 is 640 lines long, this was completely straightforward.

5.6 From maps to hypermaps

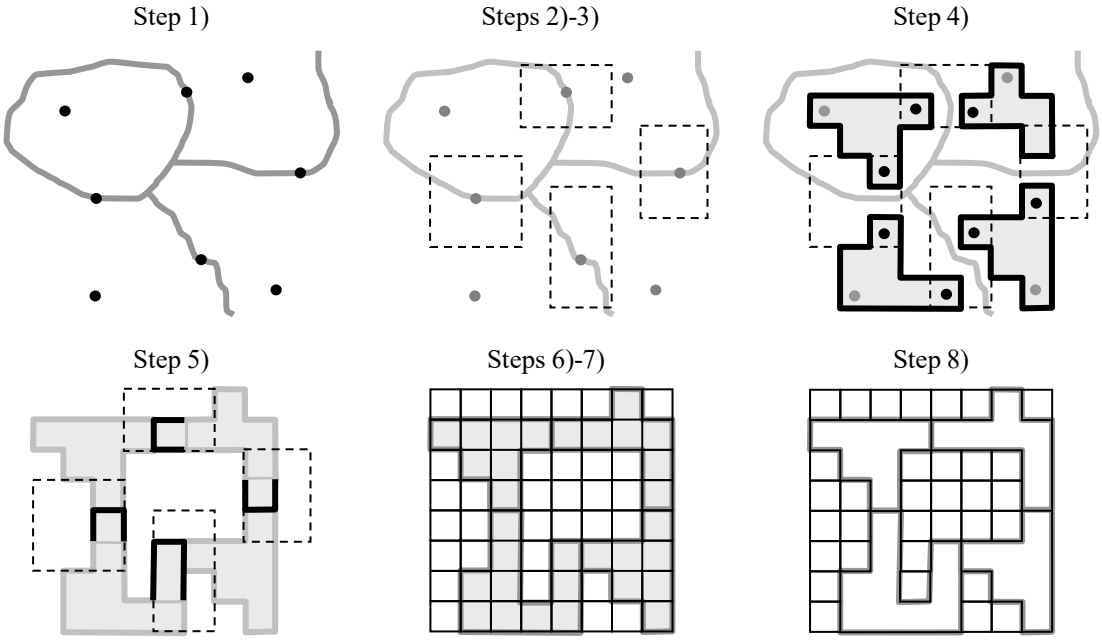
Like the Alexander proof of the Jordan Curve Theorem [25], our proof that an arbitrary map coloring problem can be reduced to a set of hypermap coloring problems uses discrete *grids* to approximate the continuous plane. Specifically, we use grids based on points with coordinates (x,y) that are binary fixed point numbers, i.e., such that $2^s x$ and $2^s y$ are both integers, for some fixed non-negative integer s (the *scale* of the grid). We approximate the open connected regions of the map with simple polygons that are the union of grid squares, which we call *mattes* (i.e., the vertices of a matte are grid points, and the sides of a matte are parallel to one of the axes).

We really only need to construct discrete approximations of *finite* maps, since it is well-known that the compactness theorem for propositional logic implies that the Four Color Theorem for infinite maps follows from the Four Color Theorem for finite maps. However, the proof of the compactness theorem relies on the axiom of choice, which we wished to avoid. Therefore, we specialized the compactness argument to the Four Color Theorem, making use of the fact the every open region contains a grid square for a suitable scale (this gives an effective enumeration of the regions of a map).

[12] Actually, a pretty-printed and somewhat more readable version of these scripts.

The discrete coloring problem for a finite map is constructed in several steps:

1) Pick, for each pair of adjacent regions, a common border point that is not a corner; also pick a point in each region.
2) Pick, for each of the border points chosen in step 1), a small rectangle that contains the point but that is disjoint from other regions and the other rectangles chosen in this step.
3) Pick grid rectangles (rectangles whose vertices are grid points) that approximate from inside the rectangles chosen in step 2); the approximation of each rectangle must contain the corresponding border point from step 1).
4) Pick mattes that approximate the map regions from the inside; the matte approximation of a region *R* must contain the point chosen in step 1) and meet the inside of every grid rectangle from step 3) that corresponds to a border point of *R*.
5) Now the interior of each border rectangle *B* meets two region mattes; call them *Q* and *R*. Extend *Q* with grid squares included in *B* until Q contains a square adjacent to a square contained in *R*; do this for each *B*.
6) Choose a grid rectangle that contains all the extended mattes from step 5) – a bounding box.
7) Construct the hypermap that corresponds to the map whose regions are the individual grid squares contained in the bounding box from step 6), and the exterior of that box.
8) Each matte from step 5) now corresponds to a simple ring cycle in the hypermap from step 7); use the converse of the "patch" operation defined in §5.1 to construct the remainder map of that ring (in essence, erasing the darts corresponding to grid segments inside the matte). Repeat for each matte; the result is the desired hypermap.



To summarize, we take the hypermap for a regular rectangular grid and punch out approximations of the map regions. It is interesting to note that in this construction, the planarity of the final hypermap is a direct consequence of the planarity of the rectangular grid hypermap. The latter is established simply by counting: the map for an $m \times n$ rectangle has $(m+1)(n+1)$ nodes, $mn+1$ faces, $m(n+1) + (m+1)n$ edges, hence

$$N + F - E = (m+1)(n+1) + (mn+1) - m(n+1) - (m+1)n = 2$$

so the Euler formula holds. The Jordan Curve Theorem plays no part in this.

In step 8) the mattes and bounding box should all use the same scale. This can obviously always be achieved by subdivision. In the Coq proof, we reduce everything to a common scale just after step 4), and perform steps 5) to 8) on the integer grid; even in steps 3) and 4), we use only rectangles and mattes on the integer grid, introducing the scale factor separately: for example, we use the following definition: an integer matte Q contains a point $P = (x,y)$ at scale s iff $(\lfloor 2^s x \rfloor, \lfloor 2^s y \rfloor) \in Q$.

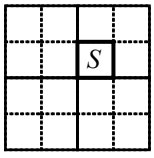
Steps 1), 2) and 3) follow from the topological definitions, classical reasoning, and the fact that grid points form a dense subset of the plane. Step 4) is carried out by choosing, for each region R of the map, an increasing sequence of mattes approximating R from inside, $Q_0 \subseteq Q_1 \ldots \subseteq Q_n \subseteq R$. We take for Q_0 a single grid square containing the point chosen in step 1). For any i, if there is some border grid rectangle B from step 3) that meets R but not Q_i, we choose some point P in $B \cap R$, and choose a matte Q_{i+1} such that $Q_i \cup \{P\} \subseteq Q_{i+1} \subseteq R$; if there is no such B then we are done (we set $n=i$). The proof that Q_{i+1} exists is the only point in the construction where we need to use topological reasoning. We show that the union $\mathfrak{Q}_i$ of all mattes Q such that $Q_i \subseteq Q \subseteq R$ is a nonempty clopen of R; since R is connected, this implies $\mathfrak{Q}_i = R$, hence $P \in \mathfrak{Q}_i$, and the existence of Q_{i+1} then follows from the definition of $\mathfrak{Q}_i$. That $\mathfrak{Q}_i$ is both closed and open in R follows from two combinatorial properties of mattes; for any matte Q and grid rectangle N,

1. For any point $P \in N$, there is a matte Q' such that $Q \cup \{P\} \subseteq Q' \subseteq Q \cup N$.
2. For any point $P \in N \cap Q$ there is a matte Q', such that $Q \subseteq Q' \subseteq Q \cup N$ and Q' contains all grid squares adjacent to P.

Property 1 implies that $\mathfrak{Q}_i$ is closed, and 2 implies that is $\mathfrak{Q}_i$ open. Both properties are immediate consequences of a more specific property of integer mattes

3. For any matte Q whose vertices have *even* integer coordinates, any rectangle N that meets Q and whose vertices have integer coordinates, and any unit square S included in N and not touching the border of N, there is a sequence $S_0 \ldots S_n = S$ of unit squares of N such that $S_0 \subseteq Q$, and $Q \cup S_1 \cup \ldots \cup S_i$ is a matte for all $i \leq n$.

In fact, property 3 is the only significant combinatorial property we need to prove on mattes, because it also provides the means of carrying out step 5) of the construction. Property 3 is nontrivial because of the restriction that mattes must be *simple* polygons; that restriction is needed in step 8). This restriction prevents us from blindly adding all the unit squares in N to Q: we have to carefully select the sequence $S_0 \ldots S_n$ so that it stays away from Q after S_0 and S_1. This is possible because of the assumption that the vertices of Q have even coordinates. Consider the 4×4 square with even coordinates that covers S and all the unit squares adjacent to S. The assumption on Q implies that each of the four 2×2 quadrants of the 4×4 square is either contained in or disjoint from Q.



If S is contained in Q we can take n=0; if not, the 2×2 quadrant that contains S is disjoint from Q. If one or two of the quadrants adjacent to S meet Q then they are contained in

Q so we can take n=1, letting S_0 be the square of that quadrant adjacent to S. If Q meets only the quadrant that touches S diagonally, then we can take n=2, letting S_0 be the square of that quadrant that touches S (diagonally), and S_1 be either of the two squares adjacent to both S_0 and S. Finally, if the entire 4×4 square is disjoint from Q then by induction[13] on the size of N there is a sequence of unit squares that extends Q, ends with a unit square S' adjacent to S, and lies entirely on one side of the line separating S from S', i.e., if S' is just below S, then the sequence contains only squares below S. In that last case we can simply append S to the sequence.

Finally, step 6) is a straightforward computation, and step 7) is just a matter of coining arithmetic formulas for the e, n, and f permutations, given the following system of coordinates:

- grid points have integer coordinates
- grid unit squares have the same coordinates as their lower left-hand corner
- darts of the hypermap are unit vectors between two grid points. The coordinates of a dart are obtained by adding the coordinates of its origin to the coordinates of the unit square that lies counter clockwise of the dart.

The dart coordinates enjoy several useful identities. A dart with coordinates d belongs to a square with coordinates $\lfloor d/2 \rfloor$, has origin d-$\lfloor d/2 \rfloor$, and its orientation is given by the remainder of that division (e.g., (0,0) for left-to-right). The f (resp. n) permutation simply cycles through the four remainder values while keeping the square (resp. origin) coordinates unchanged. In the Coq script these coordinates and their properties are used throughout this part of the proof; for example, they are used to carry out the complex matte extension proof outlined above in an orientation-independent way.

6 Looking back

This was a project that started out with a modest objective – trying out the internal execution engine of Coq on a "real" benchmark – but became more ambitious over time, as each successfully reached objective revealed that a new one might just be within reach.

We had come across the Robertson *et al*. proof [26] while looking for programming project for students in a basic computer science course. We wondered how a "modern" approach to the reducibility computation, based on decision diagrams, would fare against the idiomatic array-based C program supplied by Robertson *et al*., so we tried rolling out our own, in OCaml. The C program turned out to be faster, but not by much; however, we noted that the usually crucial caching and hash-consing optimizations had little impact for this problem. This meant that the reducibility computation could be carried out efficiently by a purely functional program – such as an internal Coq program.

This looked like a nice benchmark for Coq or other provers, so we decided to give it a try. We decided to start modestly by just enumerating the colorings of a configuration,

[13] In the formal proof, the induction comes before the case analysis on S and its neighbours; furthermore in the induction hypothesis we need to make a distinction between the rectangle N in which the S_i sequence is taken, and a larger rectangle N_0 that contains all the neighbours of S, but is such that Q and N_0-N are disjoint.

using explicit adjacency lists. Our program[14] was reasonably fast, even though it made liberal use of higher-order iterators, so we tried generating the decision tree, primarily to check memory requirements. The experiment showed that:

- The full computation would barely fit in memory, using Coq's internal data representation. The lower layers of the chromogram tree would have to be compressed, and this would complicate the algorithm enough to make a correctness proof worthwhile.
- Proving the correctness of such purely combinatorial programs was surprisingly easy in Coq, using the generate-and-test approach outlined in section 4. For example, a somewhat tricky function that computed in a single pass the normalized edge-coloring of a ring from one of its face colorings was harder to program than to prove correct!

These observations prompted us to switch to a more ambitious goal: a fully checked formal proof of the (computation-intensive) reducibility part of the Four Color Theorem. We enlisted the help of Benjamin Werner, and divided the work: we would port and verify the decision diagram-based check, and he would formalize the Kempe-chain proof of the validity of the "consistent" relation presented in section 5.2.

Our "program check" part turned out to be easier than expected, once we had hammered out a crude first prototype of the tactic shell described in section 4 to organize the bookkeeping that cluttered our scripts. Having finished early, we took a look at the bigger picture – which definition of planar graphs could plausibly be used to actually do a full proof of the full Four Color Theorem in Coq? It appeared that an explicit construction program definition would work best for the reducibility part. However, using this definition for the general statement seemed contrived. In particular, the proof of correctness of the contract validity check clearly depended on the Birkhoff Theorem and hence some form of the Jordan Curve Theorem, which seemed difficult to derive by induction on the construction program[15]. The line of reasoning exposed in 5.1 lead us to hypermaps, and we decided to try to validate the concept by proving the equivalence between the Euler formula and the Jordan property.

This was much harder than anticipated, because of the gap between the intuitive, picture-rich proof outline, and the very precise logical statement that had to be fed to the Coq proof assistant. We went through several styles of definition for the basic notions, such as paths. At the outset, the "Boolean predicate with reflection" approach, outlined in section 4, emerged as a clear winner. Our tactic shell proved to be surprisingly effective in this setting; in particular, we could usually derive an induction hypothesis directly from a goal with a single command, so we rarely needed spell out such hypotheses. With the addition of occurrence selection to our tactic shell, this even turned out to be nearly always true.

With the formal setting in place, we decided to tackle the contract validity check, a corner of the reducibility proof that we had originally planned to cut. The correctness of the check follows easily from the Birkhoff lemma (this is one of the cleverest innovations of the Robertson *et al*. proof). Proving the Birkhoff lemma was enticing

[14] Sadly, this program does not appear in the final version of the development. It was scrapped when we changed the representation of configurations to the explicit construction programs presented in 5.3.
[15] Other efforts at formalizing graph theory had stumbled upon this difficulty.

because it would be an example of using the reflected reducibility computation to prove a tangible graph theoretic result.

However, the proof of the Birkhoff Lemma required the graphically obvious operations of cutting up a map along a cycle and conversely of gluing back two map pieces. Defining these operations formally, and proving such intuitively obvious properties as "the whole map is planar if and only if the two pieces are" turned out to be much harder than expected, again because of the gap between pictorial intuition and computerized formalism. Indeed, the asymmetric "patch" relation described in section 5.1 emerged directly from the formal statement of the Jordan property – it seems completely counterintuitive when one only looks at plain maps or graphs. This "patch" relation is one of the enabling concepts of the formal proof; as the proofs of most of the key embedding lemmas depend on its precise formulation and properties (see section 5.4, steps 1 and 4). The "obvious" symmetry properties of plain map cut-outs in a plain map are actually not so trivial to prove formally (see file `revsnip.v`).

At this point we needed to integrate in our setting Benjamin Werner's work on the correctness of the consistency relation between edge colorings and chromograms, in order to be able to use the reflected reducibility computation in the proof of the Birkhoff lemma. The work wasn't quite complete; some lengthy case analyses were unfinished. Our own experience suggested that these manual analyses could be avoided by replacing the relational characterization of the consistency predicate with an effective functional one, and replacing lemmas that asserted the existence of chromograms with explicit functions for computing them. Indeed, it took us less than a day to write these functions and prove their correctness (using our tactic shell), and only another day to complete the entire proof, reusing lemmas from the Jordan ↔ Euler proof to build on-the-fly the construction program he needed to postulate. The entire development fits in a single file (`kempe.v`).

We also needed to relate the adjacency list data used by the reducibility check functions to our hypermap model. Extracting geometrical relations directly from the adjacency data turned out to be complex and impractical, so we decided to exploit the construction program structure that was built into the Robertson *et al*. data (it was used in their coloring function), and to generate both the adjacency data and a dependently-typed hypermap from it. The correspondence proofs were feasible, but not very elegant; they used a large number of ad hoc arithmetic manipulations, as the maps dart were defined as pairs of indices in the construction.

We were now faced with a software engineering problem: we needed to integrate formal theories whose development had spanned several years. This meant that while their main results were still usable (after all, this was a formal development), their proofs tended to rely on slightly different collections of facts about basic data structures such as integers, lists, and paths. We might have been able to complete the Birkhoff lemma proof, but a major refactoring was clearly needed for any further progress. This actually took several (part-time) months to complete.

With the cleaned-up basis in place, the Birkhoff Lemma and its immediate consequences (e.g., that any face should have at least five sides) followed easily (the "immediate consequences" actually required more work). The last bit of graph theory that remained in the Four Color Theorem proof was the construction of the embedding.

It was only at this point, when we examined how we could formalize the unpublished proof of correctness [27] of the configuration search heuristic used by Robertson *et al*., that we realized how much the unavoidability proof could be simplified, and that a *complete* formal proof of the Four Color Theorem was close at hand.

After devising the quiz compilation algorithm, we realized that we could also compile the contract map colorings, using extended construction programs as an intermediate representation. That was the death knell for the adjacency list representation and the mess of index calculations it required. We completely removed it from the proof, along with the graph coloring procedure that had been the first step of the proof.

After this, completing the combinatorial proof was mostly a (purely functional) programming exercise, described in section 5.5, and refactoring (packaging sets of geometrical assumptions in records, which we had deferred, since we didn't know which exactly sets would be useful).

Finally, the first part of the proof – the discretization of the continuous problem – was the last one to be completed. We wanted to avoid getting heavily involved in real analysis and the proof of the Jordan Curve Theorem, so we had planned to settle for an informal proof of this reduction. While writing the first draft of this paper we realized both that we would have a much stronger result if we completed this last part, and that it was possible to do this while relying mostly in the combinatorial methods we had developed for the rest of the proof – so we went ahead and did it.

7 Looking ahead

As with most formal developments of classical mathematical results, the most interesting aspect of our work is not the result we achieved, but how we achieved it. We believe that our success was largely due to the fact that we approached the Four Color Theorem mainly as a *programming* problem, rather than a *formalization* problem. We were not trying to replicate a precise, near-formal, mathematical text. Even though we did use as much of the work of Robertson *et al.* as we could, especially their combinatorial analysis, most of the proofs are largely our own.

Most of these arguments follow the generate-and-test pattern exposed in section 4. We formalized most properties as computable predicates, and consequently most of our proof scripts consisted in verifying some particular combination of outcomes by a controlled stepping of the execution of these predicates. In many respects, these proof scripts are closer to debugger or testing scripts than to mathematical texts. Of course, this approach was heavily influenced by our starting point, the proof of correctness of the graph coloring function. We found that this programs-as-proof style was effective on this first problem, so we devised a modest set of tools (our tactic shell) to support it, and carried on with it, generalizing its use to the rest of the proof. Perhaps surprisingly, this worked, and allowed us to single-handedly make progress, even solving sub problems that had stumped our colleagues using a more orthodox approach.

We believe it is quite significant that such a simple-minded strategy succeeded on a "higher mathematics" problem of the scale of the Four Color Theorem. Clearly, this is the most important conclusion one should draw from this work. The tool we used to support this strategy, namely our tactic shell, does not rely on sophisticated technology

of any kind, so it should be relatively easy to port to other proof assistants (including the newer Coq). However, while the tactic shell design might be the most obvious by-product of our work, we believe that it should have wider implications on the interface design of proof assistants. If, as this works seems to indicate, the "programming" approach to theorem proving is more effective than a traditional "mathematical" approach, and given that most of the motivated users of poof assistants have a computer science background and try to solve computer-related problems, would it not make interface of a proof assistant more similar to a program development environment, rather than strive to imitate the appearance of mathematical texts?

Acknowledgements

The work of Benjamin Werner on reducibility correctness was an integral part of this development, even though it does not appear in the final proof script; we benefited from extended discussions with him throughout this work, several of which also involved Vincent Danos. We thank Kenneth Appel for a historical perspective on the original Four Color Theorem proof, Robert Cori for pointing out the related work on hypermaps, and Hugo Herbelin and Bruno Barras for always patiently helping us (mis)use the Coq assistant. We are also grateful to Cédric Fournet for his careful proofreading of the initial draft of this report.

Rerefences

1. S. B. Akers, 'Binary Decision Diagrams', *IEEE Transactions on Computers* **c-27**(6) (1978), 509–16.
2. T. Altenkirch, 'Constructions, Inductive types and Strong Normalization', PhD Thesis, University of Edinburgh, 1993.
3. K. Appel and W. Haken, 'Every Planar Map is Four Colorable', *Bulletin of the American Mathematical Society* **82** (1976), 711–12.
4. K. Appel and W. Haken, 'Every Planar Map is Four Colorable, Part I: Discharging', *Illinois Journal of Mathematics* **21** (1977), 429–90.
5. K. Appel and W. Haken, 'Every Planar Map is Four Colorable, Part II: Reducibility', *Illinois Journal of Mathematics* **21** (1977), 491–567.
6. K. Appel and W. Haken, *Every Planar Map is Four-Colorable*, Contemporary Mathematics **98**, American Mathematical Society, 1989.
7. D. Aspinall, 'Proof General: A Generic Tool for Proof Development', *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, (TACAS) 2000*, Springer-Verlag LNCS 1785, 2000.
8. G. Barthe, J. Hatcliff and M.H. Sørensen, 'CPS translations and applications: the Cube and beyond', *Higher Order and Symbolic Computation* **12**(2) (1999), 125–70.
9. Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Construction*, Texts in Theoretical Computer Science, an EATCS series. Springer Verlag, 2004.
10. G. D. Birkhoff, 'The reducibility of maps', *American Journal of Mathematics* **35** (1913), 115–28.
11. S. Boutin, 'Using reflection to build efficient and certified decision procedures', *Proceedings of Theoretical Aspects of Computer Science (TACS) 1997*, Springer-Verlag LNCS 1281, 1997.

12. R. E. Bryant, 'Graph-Based Algorithms for Boolean Function Manipulation', *IEEE Transactions on Computers* **c-35**(8) (1986), 677–91.
13. The Coq Development Team, 'The Coq reference manual, version 7.3.1', system and documentation available from ftp://ftp.inria.fr/INRIA/coq/V.7.3.1.
14. T. Coquand and G. Huet, 'The Calculus of Constructions', *Information and Computation* **76**(2/3) (1988), 95–120.
15. R. Cori, 'Un code pour les graphes planaires et ses applications' *Astérisque* **27** (1975), 169.
16. R. Cori and A. Machí, 'Maps, hypermaps and their automorphisms: a survey I, II, III', *Expositiones Mathematicae* **10**(5) (1992), 403–67.
17. H. Geuvers, R. Pollack, F. Wiedijk and J. Zwanenburg, 'A Constructive Algebraic Hierarchy in Coq', *Journal of Symbolic Computation* **34**(4) (2002) 271–86.
18. H. Heesch, *Untersuchungen zum Vierfarbenproblem*, 810/a/b, Bibliographishes Institut, Mannheim-Wien-Zürich, 1969.
19. M. Hofmann and T. Streicher, 'The groupoid interpretation of type theory', *Proceedings of Twenty-five years of constructive type theory*, Oxford University Press, 1998.
20. J. E. Hopcroft and R. E. Tarjan, 'Efficient planarity testing', *Journal of the Association for Computer Machinery* **21** (1974), 145–54.
21. G. Huet, 'The Zipper', *Journal of Functional Programming* **7**(5) (1997), 549–54.
22. A. B. Kempe, 'On the geographical problem of the four colours', *American Journal of Mathematics* **2** (part 3) (1879), 193–200.
23. K. Kuratowski, 'Sur le problème des courbes gauches en topologie', *Fundamenta Mathematicae* **15** (1930), 271–83.
24. A. Miquel, 'A Model for Impredicative Type Systems, Universes, Intersection Types and Subtyping', *Proceedings of the fifteenth annual IEEE symposium on Logic in Computer Science* (2000).
25. M. H. A. Newman, *Elements of the topology of plane sets of points*, Dover Publications (1992).
26. N. Robertson, D. Sanders, P. Seymour, and R. Thomas. 'The Four-Colour Theorem', *Journal Combinatorial Theory, Series B* **70** (1997), 2–44.
27. N. Robertson, D. Sanders, P. Seymour, and R. Thomas. 'Discharging Cartwheels', unpublished manuscript, available at ftp://ftp.math.gatech.edu/pub/users/thomas/fcdir/discharge.ps.
28. T. L. Saaty and P. C. Kainen, *The Four-Color Problem: Assaults and Conquest*, McGraw-Hill, 1977.
29. S. Stahl, 'A Combinatorial Analog of the Jordan Curve Theorem', *Journal of Combinatorial Theory, Series B* **35** (1983), 28–38.
30. P. G. Tait, 'Note on a theorem in the geometry of position', *Transactions of the Royal Society of Edinburgh* **29** (1880), 657–60.
31. W. Tutte, 'Duality and trinity', *Colloquium Mathematical Society Janos Bolyai* **10** (1975) 1459–72.
32. W. T. Tutte, 'Combinatorial oriented maps', *Canadian Journal of Mathematics* **31** (1979), 986–1004.
33. T. R. S. Walsh, 'Hypermaps versus bipartite maps', *Journal of Combinatorial Theory, Series B* **18** (1975) 155–63.
34. D. W. Walkup, 'How many ways can a permutation be factored into two *n*-cycles?', *Discrete Mathematics* **28** (1979), 315–19.
35. B. Werner, 'Une Théorie des Constructions Inductives', PhD Thesis, Université Paris VII, 1994.

36. R. Wilson, *Four Colors Suffice*, Penguin books, 2002.