Fast and Correct Round Elimination

Joonatan Saarhelo

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 2022

Supervisor and advisor

Prof. Jukka Suomela

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Joonatan Saarhelo

Title Fast and Correct Round Elimination

Degree programme Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor and advisor Prof. Jukka Suomela

Date 2022 **Number of pages** 51 **Language** English

Abstract

Round elimination is a process used to study the hardness of distributed graph problems. It is available as a computer program. Optimizations to the program have permitted the study of more complex problems. Because no fast algorithm for checking the result of round elimination is known, it must be implemented correctly. In this thesis I present a new algorithm for maximization, the most computationally expensive part of the original Round Eliminator, prove that the new algorithm is asymptotically faster and formally verify the theory behind it in Coq along with a program that checks maximization results using the new algorithm. An implementation based on the new algorithm makes round elimination orders of magnitude faster in practice. Unfortunately the verified checker is too slow for practical use but that is not a fundamental limitation, only a flaw in the particular way it was written.

Keywords round elimination, Coq, distributed algorithms, formal proof

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Joonatan Saarhelo

Työn nimi Nopea ja virheetön kierroseliminaatio

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja ja ohjaaja Prof. Jukka Suomela

Päivämäärä 2022 **Sivumäärä** 51 **Kieli** Englanti

Tiivistelmä

Kierroseliminaatio on menetelmä, jota käytetään hajautettujen verkko-ongelmien aikavaativuuden tutkimiseen. Siitä tehdyn tietokonetoteutuksen optimoiminen on mahdollistanut yhä mutkikkaampien ongelmien tutkimisen. Koska kierroseliminaation tuloksen tarkistamiseen ei tunneta nopeaa algoritmia, on hyvin tärkeää että kierroseliminaatio on toteutettu oikein. Esittelen tässä työssä uuden algoritmin kierroseliminaation hitaimpaan osaan, maksimointiin. Osoitan, että uuden algoritmin aikavaativuus on vanhaa parempi ja todistan Coqilla teorian johon se perustuu sekä ohjelman kierroseliminaation tuloksen tarkistamiseen. Käytännössä uutta algoritmia hyödyntävä toteutus kierroseliminaatiosta on yli tuhat kertaa vanhaa nopeampi. Valitettavasti todistettu kierroseliminaation tarkistaja on liian hidas ollakseen hyödyllinen, mutta tämä vika on korjattavissa.

Avainsanat kierroseliminaatio, Coq, hajautettu laskenta, matemaattinen todistus

Contents

Symbols and abbreviations

Symbols

Δ	graph maximum degree
Π_n	round elimination problem
Σ_n	alphabet of Π_n
A_n	active side configurations of Π_n
P_n	passive side configurations of Π_n

Abbreviations

RE	round elimination
GADT	generalized algebraic data type
ATP	automatic theorem prover
ITP	interactive theorem prover

1 Introduction

Today's world is filled with software and most of it does not always work as intended. The consequences range from mere annoyances to expensive or life-threatening incidents.

Programming languages that completely eliminate invalid memory accesses or even the bulk of crashes have entered the mainstream. Yet all mainstream languages are Turing-complete and as a consequence allow writing programs that hang indefinitely. Programs that not only do not get stuck but even behave as intended are so rare that there is a large body of research that tries to predict where the bugs are [30].

At the same time, computers have become an integral part of mathematical research. This is rather surprising – while mathematicians have published many mistakes [22], most programs contain several. Still, computer programs are good for testing conjectures on millions of different inputs in the hope of finding a small counterexample that is easy to check by hand. But not all outputs are easy to check.

Round elimination (RE) is a process used in proofs about the time complexity of distributed algorithms [5]. No better algorithm for checking the result of RE than performing RE is known, so the only way to produce reliable results is to have a reliable implementation. (This thesis introduces an algorithm for checking RE that saves some work but involves the same operations as performing it, so implementing it is not significantly easier than implementing full RE.)

Round elimination can be tedious or downright impossible to perform by hand, as it may have to be applied multiple times in succession, with every application multiplying the size of the working set. Faster algorithms have allowed RE to be used on a wider variety of problems, so speeding it up is desirable.

Hence, the goal of this thesis is to produce a faster but also trustworthy implementation of round elimination. I tackle only one part of RE, maximization, as that part accounts for most of the computation time and is easy to specify.

Section 4 explains how to perform round elimination and how it helps in analyzing the class of problems introduced in Section 3. Section 5 is a proof sketch derived from the mechanized proof discussed in Section 7, which I wrote for the new maximization algorithm that I developed. Finally, Section 8 compares the asymptotic performance of the new algorithm to the state of the art.

2 Background

This section goes over different methods of using computer programs to check software correctness with a focus on theorem proving. For background on round elimination, see Section 3 and Section 4.

2.1 Conventional methods of improving correctness

Testing can in fact be used to show the absence of bugs, but only in programs that take a small number of different inputs. On the other hand, such a program is only as powerful as a look-up table.

Fuzzing is the practice of running software against computer-generated inputs to find crashes. AFL-fuzz [2] is a highly optimized fuzzer that monitors program execution in order to find inputs that trigger previously unexplored execution paths. It has been used to find intricate bugs in complex software. One crash that I have found via fuzzing was caused by code that assumed that one can compute the absolute value of any machine integer. However, taking the absolute value of the smallest representable integer (for example -128 for 8-bit integers) leads to undefined behaviour because there is always one negative integer more than there are positive integers in two's complement.

Because of the program instrumentation, fuzzing is able to find interesting inputs even though it cannot go through the whole input space. It optimizes code coverage of the executable rather than the source code. Fuzzing is best at finding crashes but it can be used to find incorrect behaviour by writing a program that crashes whenever an incorrect output is produced. However, that is only effective if outputs can be checked very quickly. Fuzzing is of little use in the case of round elimination because RE is very slow to perform and there is no way to check the result except by running round elimination on it. At best, one could compare a new algorithm to a simpler but even slower implementation of RE.

SAT solvers are programs that are optimized for solving the Boolean satisfiability problem. They can be used to find cases where formulas fail on finite but relatively large inputs and even perform an exhaustive search to show that no such cases exist. The biggest downside is that turning algoriths into Boolean formulas can be challenging.

Just like fuzzers, SAT solvers suffer from the problem that round elimination is difficult to check. If it was easy to check, one could just ask a SAT solver to find a solution. I actually did that with Z3 but as expected, the performance was bad because the Boolean encoding of what a correct result of maximization is is very large.

It seems that the more popular methods focus on making sure that the produces a correct result most of the time, operating on the assumption that the result is easy to verify. The requirements of software for mathematical research are very different. It is completely acceptable for the software to sometimes fail, as long as it is correct when it succeeds. Given a way to check an output, the easiest way to achieve that is with a runtime assertion.

2.2 Theorem proving

A mechanized proof is a fully formal proof in the spirit of Principia Mathematica, checked by a computer. It can establish correctness of algorithms even for arbitrarily large inputs and is completely unaffected by how expensive it is to compute or check outputs, which makes it ideal for verifying round elimination. But ideally, I would like to prove that an implementation is correct, not just the algorithm.

I chose to write the implementation in Coq, which is both a programming language and a logic (see Section 6), so there was no need to somehow import the program into a logic to allow proving it. Embedding round elimination in Coq makes it possible to prove whole reasoning chains, while a verified implementation of only round elimination could be accidentally given the wrong input.

Another option is to generate lemmas from the source code that is being verified. The seL4 microkernel [20] has been developed in that fashion, for example. High performance and low level access to hardware is vital when implementing an operating system, so it is desirable to write it in C rather than a safe high-level language.

Electrolysis [32] is an interesting take on the latter. It is a direct translation from a subset of Rust to Lean 2 (a theorem prover similar to Coq) that preserves semantics but not performance. It could have worked well for me as I would have gotten a Rust and a Lean implementation of the algorithm but unfortunately version 2 of Lean is very old, as is the required version of Rust.

The most straightforward way to write the proof is to use an automatic theorem prover (ATP). But the power of ATPs is somewhat limited; that they would not be able to prove something as complex as a round elimination algorithm. The ATP's job can be made easier by splitting the proof into smaller lemmas but not all proofs can be subdivided in a productive way.

Interactive theorem provers (ITPs), on the other hand, have been used to verify contemporary mathematics [22]. In them, the user constructs the proof in a conversation with a computer program. The user indicates how to advance the proof and the program reports the new state after that.

There have been relatively successful attempts to allow the use of ATPs inside ITPs, like Hammer for Coq [13] named after the similar Sledgehammer in Isabelle/HOL. Lemmas have to be converted in a lossy process because the best ATPs operate on first order logic but ITPs use higher-order logics, as they are more palatable to humans and even necessary to encode some mathematical objects. Because of the conversion, some lemmas are successfully proved in an ATP but the proof is not accepted by the ITP. This is subject to change as the popular TPTP [28] problem set for ATPs has added problems in a typed higher-order form.

Perharps the most notable example of an ITP proof is Thomas Hales' proof of the Kepler conjecture. The Kepler conjecture states that there is no way of packing spheres more densely than the cannoball packing, in which the spheres are arranged in hexagonally packed layers and those layers are stacked on top of each other. In 1831, Gauss proved a proposition about polynomials that can be used to show that a variant of the aforementioned packing is the densest possible lattice packing of spheres [31]. However, before Hales' proof the possibility of a denser irregular

packing was not ruled out.

A proof consisting of computer programs that (among other things) solved linear programming problems and hundreds of pages of notes was completed in 1998 [22]. After four years of inspecting the proof, a panel of twelve referees announced that they couldn't be completely certain if the computer calculations were correct. That prompted Hales to start the Flyspeck project, which completed a formal proof of the Kepler conjecture written in HOL Light in 2014 and found a few bugs in the original code [22].

2.3 Proof automation

Completely formal proofs can be large and repetitive and a minute change in the theorem to be proved can change their shape. Directly constructing proofs by hand seems like a fool's errand. Instead, the bulk of proofs is produced via various forms of code generation.

Most proof assistants read a proof script from disk and generate the proofs anew on every run. The only exception that I am aware of is Metamath [7], in which a simple program checks a proof file that is not human-readable. The proof file is typically produced with untrusted tools, as only the checker needs to be correct.

The other tools prefer human-readable proofs because they are maintainable. One may wonder why a proof needs to be maintainable, as once it is complete, it does not need to be modified anymore. The reasons are the same as in software engineering in general: while the proof is in development, there may be unanticipated changes.

As an example, it may turn out that there is a data structure that is easier to work with than the one initially used. *Transporting* the old proofs to the new data structure takes great effort. On the other hand, automated proofs will often work with a different data structure with little to no editing. Besides, if the initial formalization of the mathematical object in question was inaccurate, there is no other option than to change data structures.

2.3.1 Aside: transporting proofs

It is very common in mathematics to prove something for one representation of a mathematical object and assume that it is true for all isomorphic representations. This notion, while intuitive is not trivial to formalize and current proof assistants do not support it by default.

An easy way to make a proof that applies to all representations of an object is to write a proof that applies to anything with certain properties. For example, instead of relying on a particular representation of natural numbers, one can write a proof that applies to anything satisfying the Peano axioms [34]. But that makes proofs more difficult as they cannot reap the benefits of any particular representation.

An important motivation for transporting proofs between representations is *refinement*, replacement of data structures with more efficient ones [10]. In fact, the

development discussed in this work would be far more useful were the data structures refined to to ones that are of practical use. This is elaborated on in Section 9.1.

Unlike other attempts, Univalent foundations [33] support transporting any function over equivalences. However, it is fundamentally different from the Calculus of Inductive Constructions, the underlying logic of Coq, which is why the Coq implementation involves a Univalence axiom. As axioms in Coq have no computational meaning, transported functions cannot be executed, which is not a problem for proof but renders transported executable code useless. There are newer type systems, for instance Cubical type theory [11], which assign computational meaning to univalence. Equivalences for Free [29] describes an approach that manages to transport most useful constructs while remaining in Coq.

2.3.2 Tactics languages

Interactive theorem provers typically have a separate tactics language, which is used to direct proofs. The ML family of programming languages which Coq belongs to, spawned from the tactics language of Automath [14], an early proof assistant developed by de Bruijn. For Coq, the Mtac2 plugin allows writing tactics in Coq itself rather than its tactics language Ltac [1].

Tactics languages are designed very differently from specification languages, as the correctness of tactics programs does not need to be clear to the reader. They are typically untyped and highly implicit. However, Coq's reference manual [1] reports that defining complex tactics in the Ltac language has shown its fragility. It is not clear what an ideal tactics language would look like.

2.3.3 Reflection

Procedures like solving linear programming problems can be directly incorporated into proofs but compared to a function implementing the procedure, they perform poorly and produce large proofs that consume a lot of memory. Instead, one can implement the procedure, prove it correct and use it in a proof [8].

Reflection is the process of converting a Coq term to a representation that can be manipulated in Coq itself and back. Certified Programming with Dependent Types [8] automates this with tactics but Coq's canonical structures can also be used effectively for that task [15] as outlined next.

2.3.4 Type classes

Haskell type classes and Rust traits are an example of code generation in mainstream programming languages. These features select an appropriate function depending on the arguments' types. For example, the Haskell type class Show is implemented for integers and for lists of things that implement Show. These implementations are automatically composed when printing a list of integers.

Canonical structures fullfill a similar purpose to type classes in Coq. A notable difference is that canonical instances are selected via unification. Gonthier et al. [15] exploit the fact that Coq only unfolds terms when unification fails to automatically

try multiple instances in sequence. By writing instances akin to a logic program, one can automate things like ordering terms correctly in order to apply a lemma.

I do not use this kind of automation in my code, as I only learned about it afterward, but it seems highly useful. Reordering things is easy but code that has to explicitly state the desired order of terms before applying a lemma is tedious and breaks when the reordered expression changes.

3 Locally checkable labeling

Locally checkable labeling (LCL) is a family of graph problems in which one has to assign colors to edges or vertices, satisfying some locally checkable condition. [24]

Locally checkable means that a solution can be checked by checking fixed size neighborhoods. For example, to verify that vertices are properly colored, one just has to look at the neighbors of each vertex and check that no neighbor has the same color.

When every vertex has at most Δ outgoing edges, an LCL-problem can be given as a finite set of allowed neighborhoods, where Δ is the maximum degree of the graph.

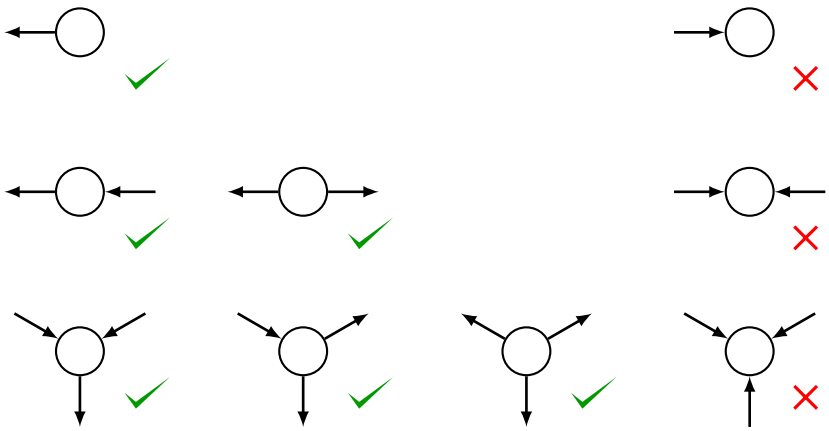


Figure 1: An exhaustive list of neighborhoods for checking sinkless orientation in a graph with maximum degree three.

I will use *sinkless orientation* as a running example. It is the problem of selecting a direction for each edge so that no vertex has only incoming edges. To check sinkless orientation, it is suffices to see the neighboring edges. There are other problems that require seeing a larger neighborhood, such as distance-2 coloring, the problem of choosing different colors for every pair of vertices the shortest distance of which is two.

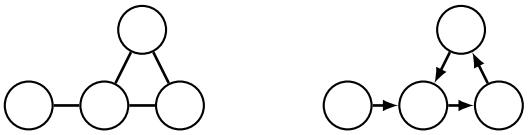


Figure 2: An undirected and the unique solution to sinkless orientation in it.

3.1 Round elimination problems

Biregular trees are trees that have properly two-colored vertices and where the degree of each class of vertices is constant. Round elimination operates on edge coloring

LCL-problems in biregular trees that can be checked by only checking the nearest edges' colors. I call this kind of problem a *round elimination problem* or RE problem.

Sinkless orientation is not such a problem. To begin with, edge orientation is not a color. However, there is an isomorphism between the solutions our sinkless orientation problem and the solutions of an RE problem, as we will see in a moment.

One of the classes of the biregular tree is called the *active side* and the other the *passive side*. The active side nodes are active in the sense that they are the computers in a network that have to choose the colors of their neighboring edges.

Since each neighborhood consists of only a vertex and a number of edges, we can simply list all multisets of allowed edge colors. They are called *configurations* and are denoted with square brackets in this thesis. There are two sets of configurations, the configurations around active nodes, A, and the ones around passive nodes, P.

An RE problem can be represented as sets A and P. The symbols in the configurations are from an alphabet Σ.

3.2 Conversion from LCL

I will now show how to convert sinkless orientation to an RE problem. The RE problem operates on a tree where the degree of active nodes corresponds to the original graph's degree and the degree of passive nodes is two. Each vertex in the original graph corresponds to an active vertex in the RE graph and edges correspond to passive vertices, as illustrated in Figure 3.

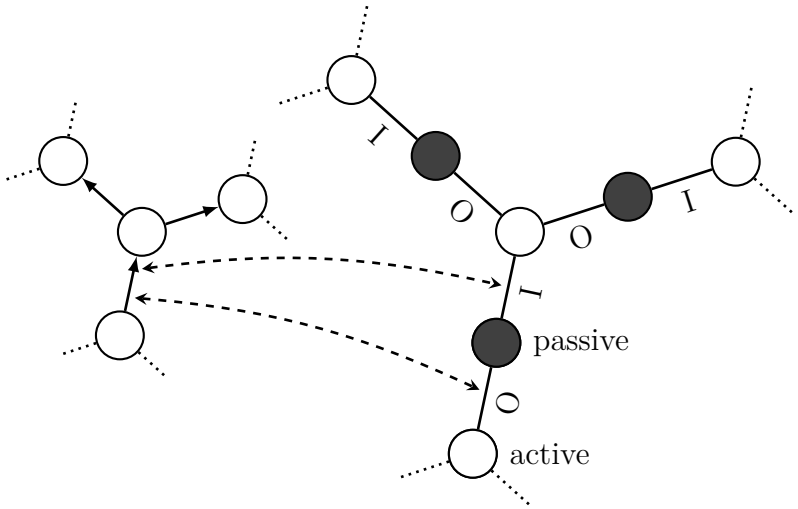


Figure 3: Correspondence between solutions in round elimination graph and original graph.

We can describe the directions of edges by coloring the RE graph's edges with I (for incoming) and O (for outgoing), so the alphabet Σ is $\{I, O\}$. The active side's rule $A = \{[I, I, O], [I, O, O], [O, O, O]\}$ lists all the valid neighborhoods. The passive side's rule $P = \{[I, O]\}$ ensures that each edge is incoming on one end and outgoing on the other; otherwise there could be edges with an arrowhead on both sides.

The cases of sinkless orientation that deal with nodes of lower degree can be disregarded, since the active nodes in the biregular tree all have the same degree. The RE problem only models sinkless orientation on regular trees.

Real-world graphs are not regular trees, since graphs without any vertices of degree one are infinite. But proofs about regular trees are still useful: if something is impossible on a regular tree, it is impossible on graphs. Some interesting graphs are almost regular trees, one just has to handle the edge cases.

The same kind of conversion as done here to sinkless orientation be applied to any locally checkable problem on regular trees. It is not always as straightforward, especially if the LCL problem looks at a big neighborhood but it is possible.

For example, distance-2 vertex coloring is quite verbose. In it, neighboring vertices have to have different colors and vertices at distance two have to have different colors. It can be encoded as follows for degree two graphs:

$$A = \{[(c_0, c_1, c_2), (c_0, c_2, c_1)] \mid \text{for all distinct } c_0, c_1 \text{ and } c_2\},$$
$$P = \{[(c_1, c_2, c_3), (c_2, c_1, c_4)] \mid \text{for all distinct } c_1, c_2, c_3 \text{ and } c_4\}.$$

Just like in sinkless orientation, the edges become passive nodes. The first value of each triplet is the color of the nearest node, the second one is the color of the neighbor the edge is going towards and the third value is the color of the other neighbor.

4 Round Elimination

Brandt et al. [5] introduced round elimination in 2019. It is a procedure that takes a round elimination problem Π_0 and outputs a new problem Π_1. It has interpretations in various models of distributed computing. As the name suggests, Π_1 can generally be solved one round faster than Π_0.

While Π_0 assigns a color to each edge, Π_1 assigns a set of colors to each edge. That set contains the colors that the edge could have in a solution to Π_0. There is uncertainty because in one round less some information that affects the solution of Π_0 is out of reach.

4.1 Lines

Dennis Olivetti [25] wrote an implementation of round elimination called Round Eliminator. Round elimination has been used to prove bounds on time complexity for various problems in the LOCAL model [4, 6, 3].

In Round Eliminator, problems are represented as *lines*. Lines are a kind of shorthand notation that compresses multiple configurations into one line. Each line is a multiset of sets and represents or *generates* all the configurations obtained by choosing one color from each set [25].

For example, the line $[\{I,O\},\{I,O\},\{O\}]$ generates the configurations $[I,O,O]$, $[I,I,O]$ and $[O,O,O]$. ($[O,I,O]$ is the same as $[I,O,O]$, since lines and configurations are unordered.)

To improve readability, all lines in this thesis are written in bold and with whitespace instead of punctuation, for instance the aforementioned line is **IO IO O**.

Figure 4: Sinkless orientation in Round Eliminator's shorthand

4.2 Formal definition

Let Π_n's alphabet, active side and passive side be Σ_n, A_n and P_n respectively. Let cfgs be a function from a line to all the configurations it generates.

Rephrased in terms of lines, the definition of RE from Distributed Algorithms 2020 [18] reads:

- Σ_1 is the powerset of Σ_0 minus the empty set,
- $A_1 = \{x \mid \forall c : \text{cfgs}(x),\ c \in P_0\}$,
- $P_1 = \{x \mid \exists c : \text{cfgs}(x),\ c \in A_0\}$.

In words: A_1 consists of all lines that only generate configurations present in P_0. P_1 consists of all lines that generate at least one configuration from A_0.

In the above equations A_1 and P_1 are defined as sets of configurations. It is strange to see $\text{cfgs}(x)$; the set of configurations generated by a configuration. That works because the configurations of Π_1 have the exact same shape as lines of Π_0: multisets of nonempty subsets of Σ_0.

4.3 Interpretation in the port numbering model

In this section I will show that round elimination produces a problem that can be solved exactly one round faster than its predecessor in the port numbering model. The port numbering model is the weakest of a number of models of distributed computing. But round elimination has been shown to work in stronger models, like the LOCAL model as well [3].

4.3.1 The port numbering model

In the port numbering model, each vertex runs the same program and must produce part of a solution to a problem defined on its graph. The problem need not be locally checkable and the graph does not need to be a tree but this thesis is about round elimination problems, so everything documented here has that shape.

Computation proceeds in a series of rounds. In each round, vertices send a message to each of their neighbors. Then all vertices receive the messages simultaneously. Finally, each vertex can either output its part of the solution and stop, or continue to the next round. Before, after and between these steps, the nodes get to compute as much as they want. This process repeats until all vertices have chosen their output.

Depending on the problem, each node may also get some information when the execution starts. For instance the total number of nodes is often provided so each node can compute how long the algoritm is going to run in advance.

However, everything stated in the previous paragraphs applies equally well to the LOCAL model for example. The defining characteristic of the port numbering model is that the only thing each vertex knows is that it is connected to its n neighbors via ports numbered 1 to n.

Formally, we can define a graph in the port numbering model as a set of connections between ports.

$$\text{Port} = \text{Vertex} \times \mathbb{Z}^+$$
$$\text{Connection} = \{\mathit{Port}, \mathit{Port}\}$$

Every port must be connected to only one other port. If a node's port n is connected, ports 1 to $n-1$ must be connected as well.

An algorithm in the port numbering model can be expressed as three functions.

$$\begin{aligned} \text{send} &: \Pi_{n:\mathbb{N}}\ \text{State} \to \text{Message}^n \\ \text{recv} &: \Pi_{n:\mathbb{N}}\ \text{Message}^n \to \text{State} \to \text{State} \\ \text{result} &: \text{State} \to \text{Output} + 1 \end{aligned}$$

The first one computes what messages to send based on the state the node is in; the second how a node's state changes when it receives its neighbors' messages. The first message in the tuple produced by send is sent to port 1, the second to port 2 etc. Likewise, the first message in the tuple passed into recv is the one received from port 1. Finally, the function called result computes a node's output if it has stopped and returns unit otherwise. (The notation Output + 1 means Output or unit. The one stands for a type that is inhabited by only one value.)

Note that send and recv may behave differently based on the number of ports. In fact, they have to because send must produce a tuple of messages of the correct length. The number of ports, n cannot be a normal parameter, as the type of the function depends on its value. For an in-depth discussion of the Π notation see Section 6.3.1.

An r-round solvable problem is a problem for which there exists an algorithm that gives the correct output after r rounds. It is easier to prove that round elimination works in the port numbering model by adopting a different perspective.

Lemma 4.1. *A problem on trees is r-round solvable in the port numbering model if and only if each vertex can produce a correct output by looking at its r-neighborhood.*

Here r-neighborhood refers to the subgraph containing vertices and edges at most r edges away.

Proof. The r-neighborhood is sufficiently large, as information from further away cannot be seen in r rounds.

Every vertex can gather its r-neighborhood in r rounds, so there is not too much information either. The gathering process works as follows: Every round, every node sends through each of its ports the number of the port, its number of neighbors and the messages it received in the previous round along with any data it was initialized with.

After round r, all vertices can reconstruct their r-neighborhood. The structure of nested messages exactly mirrors the structure of the graph.

In general graphs, the neighborhood cannot be reconstructed because if two distinct paths lead to a node that looks the same, there is no way to tell if the paths go to the same node or not. But in trees there is only one path between any two vertices. □

In the LOCAL model this lemma holds for general graphs because the LOCAL model assigns a unique identifier to every node.

4.3.2 Round elimination in the port numbering model

Suppose there is an algorithm G_0 that solves the round elimination problem Π_0 in r rounds, where $r > 0$. Let us construct an algorithm G_1 that operates on the same graph but the active nodes are passive and vice versa. Also suppose that G_1 finishes in $r-1$ rounds, so it makes a decision based on an $r-1$ neighborhood.

Suppose that G_1 is run on some vertex v. For each neighboring vertex n, G_1 computes all possible r-neighborhoods centered around n. Then G_1 runs G_0 on each of the neighborhoods, noting the color G_0 assigns to the edge $\{v, n\}$. G_1 assigns the union of all those colors to $\{v, n\}$. Figure 5 shows an example of the neighborhoods.

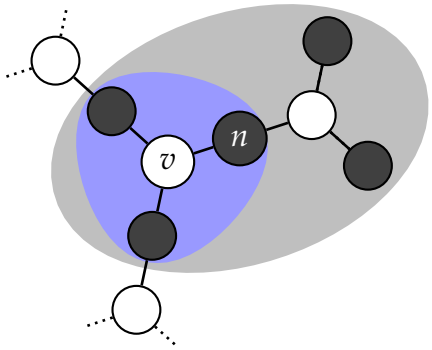


Figure 5: When $r = 2$, G_0 running on node n would see the light grey area but v running on G_1 only sees the blue area, leaving a significant blind spot.

Lemma 4.2. *G_1 solves Π_1, the result of running round elimination on Π_0, provided that disjoint parts of the graph are independent.*

Proof. It suffices to show that G_1's outputs satisfy Π_1's active and passive side rule.

All the edge colors surrounding a passive node are computed from the same part of the graph but with different blind spots. Since G_0 is run with all possible values in the blind spots, there is a run where the blind spot has the same values as it does in reality. Thus each edge's set contains the color that G_0 would assign to that edge. Because G_0 solves Π_0, A_0 must contain a configuration with those colors. Therefore P_1 is satisfied, as all the passive nodes' surrounding edges generate a configuration from A_0.

The edges $e_1, e_2, \ldots$ surrounding an active node are computed from neighborhoods whose blind spots are completely disjoint since the graph is a tree. Suppose the edges's colors generate some configuration $[a_1, a_2, \ldots]$ that is not in P_0. Then w.l.o.g. there exist some values for the blind spots such that G_0 colors e_1 with a_1, e_2 with a_2, etc. As the blind spots are completely independent, it is possible to set them all to those specific values at the same time, which results in a graph where G_0 colors the edges around a passive node with a set of colors that is not found in P_0 which is a contradiction, as we assumed that G_0 solves Π_0. Thus we know that the colorings of active nodes only generate configurations that are in P_0. □

Moving to the LOCAL model violates the independency requirement, as two blind spots cannot have the same unique identifiers. Another way to break independency is to choose a very large r. If r is so large that it could contain the whole network, the size of the network could prevent the largest options for the blind spots from existing at the same time.

Lemma 4.3. Π_1 *can be solved at least one round faster than* Π_0.

Proof. Suppose that G_0 is the fastest possible algorithm that solves Π_0. By definition, G_1 solves Π_1 one round faster than G_0. □

Lemma 4.4. Π_1 *can be solved at most one round faster than* Π_0.

Proof. If there is an algorithm G_1 that solves Π_1 in less than $r-1$ rounds, it is possible to solve Π_0 in less than r rounds: run G_1 on all passive nodes and send the resulting edge colors to the neighboring active nodes. The definition of P_1 guarantees that each active node is able to pick a configuration that is in A_0 from the edge colors around it. The definition of A_1 guarantees that the passive rule is satisfied no matter what color the active nodes pick.

But we assumed that the fastest algorithm solves Π_0 in r rounds, so the assumption that Π_1 can be solved in less than $r-1$ rounds must be false. □

We have seen that Π_1 can be solved at least one round faster and at most one round faster than Π_0. It follows that it can be solved exactly one round faster than Π_0.

4.4 Optimizing output size

Round elimination produces a gigantic number of configurations. Especially the new passive side contains almost every possible configuration. For example, suppose $\Sigma_0 = \{a, b, c\}$ and A_0 has a configuration $[a, b]$. Now P_1 contains **a b** but also **ab b**, **ac b**, **abc b**, **a ab**, **a bc**, **a abc**, **ab ab**, **ab bc**, **ab abc**, **ac ab**, **ac bc**, **ac abc**, **abc ab**, **abc bc** and **abc abc**. If we were to use this directly, the problem size would grow at an alarming rate with successive applications of round elimination, so it makes sense to optimize output size over round elimination speed.

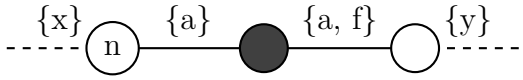
One simple optimization is to discard all active configurations containing symbols that do not appear in any passive configurations and vice versa, as using them is impossible. But even more can be discarded.

As mentioned earlier, the lines of Π_0 are just like the configurations of Π_1. (Which is why I am using line notation to write the latter as well.) In terms of lines, A_1 is the set of all *valid* lines w.r.t. P_0, where valid lines are lines that only generate configurations that are in P_0.

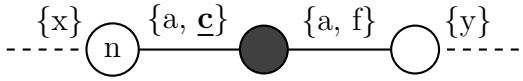
I will now show that using only some valid lines as A_1 instead of all valid lines creates a problem has strictly less solutions than Π_1 but is exactly as hard.

4.4.1 Domination

Suppose the picture below is part of a solution to Π_1. One can see that **x a** and **af y** are in A_1 and **a af** is in P_1.



If **x ac** is in A_1, there is also a solution where node n chooses it instead of **x a**, provided that **ac af** is in P_1. As **a af** is in P_1, it generates some configuration from A_0. Because **ac af** generates strictly more configurations it is in P_1, too.



As every solution using **x a** can use **x ac** instead, **x a** can be omitted.

Any configuration C in A_1 can be omitted if there is a B in A_1 that can be paired up with C so that for each pair (c, b) where $c \in C$ and $b \in B$ it holds that $c \subseteq b$ [18]. For any C and B satisfying that criterion I say that B *dominates* C and C *is dominated by* B.

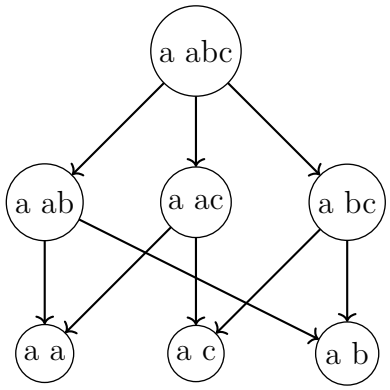


Figure 6: Graph of lines strictly dominated by **a abc**.

Note that domination is not the same relation as the subset relation of generated configurations. The line **a abc ade** generates $[a, b, e]$, while **a a bcde** only generates configurations that are in the former. Yet they are incomparable in terms of domination.

4.4.2 Maximal form

We do not need to include dominated lines in A_1, as a good algorithm solving Π_1 would never choose them, as the more dominant lines are always a better choice.

Thus the only lines needed in A_1 (the *maximal form* of P_0) are the valid lines that no valid line strictly dominates. In other words, the *maximal form* of S is the Pareto front of *valid* lines with respect to domination.

This work's main contribution is a new algorithm for finding the maximal form of P_0 by combining lines and a proof of its correctness.

4.5 Aside: Line uniqueness

Most sets of configurations can be represented by many different sets of lines but every line uniquely identifies a set of configurations. That is, there are no two lines that generate the same set of configurations.

I will show that there is only one way to turn configurations generated by a line back into a line.

Every color must be present in the line as many times as in the configuration that contains the highest number of it. Less obviously would not work and if it was present more times, there would be a configuration containing it that many times.

Using the information from the previous paragraph, it is possible to figure out the overlap between any two colors a and b. If there is no configuration with all as and all bs, one set in the line must have both. The magnitude of the overlap is the difference of how many as and bs there are and the highest number of a and b seen at the same time.

The amount of overlap tells us how many sets there are which contain both a and b. But some of those sets could also contain c. To figure out how many sets contain a, b and c, we first figure out how many contains a and c and how many contain b and c. That way we get a prediction for how many of all three are seen at the same time. The difference between the prediction and the observed value is how many sets contain all three.

Let o be a function from sets of colors to the maximum number of them that is observed at once in a configuration and let s be a function from sets of colors to the number of them or a superset present in the line.

$$\begin{aligned} o(\{a\}) &= s(\{a\}) \\ s(\{a,b\}) &= s(\{a\}) + s(\{b\}) - o(\{a,b\}) \\ s(\{a,b,c\}) &= s(\{a,b\}) + s(\{a,c\}) + s(\{b,c\}) - o(\{a,b,c\}) \\ &\dots \end{aligned}$$

Now the line can be reconstructed by repeatedly adding the largest set for which s is nonzero and subtracting one from all values of s associated with a subset of it. Any deviation from the process would yield a line which does not produce the same configurations, so the line is unique.

4.6 Putting it all together

To perform round elimination, one first finds the maximal form of P_0. The configurations of the new active side, A_1, are exactly that maximal form.

However, we want lines, not configurations, so the colors in each configuration get wrapped into a singleton set. For example, the Π_1-configuration $[\{a,b\},\{c\},\{a,c\}]$ turns into the Π_1-line $[\{\{a,b\}\},\{\{c\}\},\{\{a,c\}\}]$. Written using the notation for lines I use elsewhere it is **{a, b} {c} {a, c}**.

The new alphabet, Σ_1 consists of all colors found in A_1.

The new passive side is built by replacing every set in the lines representing A_0 with all colors from Σ_1 that it intersects. For instance if $\Sigma_1 = \{\{b\}, \{c\}, \{a, b\}, \{a, b, c\}\}$, $\{a, b\}$ would be replaced with $\{\{b\}, \{a, b\}, \{a, b, c\}\}$.

Finally, to make the representation more compact, Σ_1 is converted from subsets of Σ_0 to ordinals simply by enumerating the colors.

5 Maximization via line combination

In this section I will present a maximization algorithm based on combining lines and prove its correctness. The lemmas closely follow ones I have proved in Coq. The details only relevant to the proof in Coq are discussed in Section 7.

5.1 Combining lines

The new maximization algorithm mostly consists of combining lines. Two lines can be *combined* by pairing up their sets and taking the union of one pair and the intersection of the rest as illustrated in Figure 7.

$$\{I, O\} \cup \{O\} = \{I, O\}$$
$$\{I, O\} \cap \{I, O\} = \{I, O\}$$
$$\{O\} \cap \{I, O\} = \{O\}$$

Figure 7: one possible way to combine **IO IO O** with itself.

Theorem 5.1 (Combination is sound)**.** *A combination C of lines A and B only generates configurations present in A or B.*

Proof. For each configuration c generated by C, w.l.o.g. suppose the symbol chosen from the union is from A. The symbols that come from intersections are in both A and B. Thus all symbols in c are from A. □

5.2 Domination relation

A *maximal line* is a line that no valid line strictly dominates. The *maximal form* is simply a collection of all the maximal lines.

Lemma 5.1. *The domination relation is transitive.*

Proof. Let line a dominate b and b dominate c. Choose some ordering for b. Order a so that each of its sets includes b's set. Order c so that each of its sets is included in b's set. As set inclusion is transitive, the corresponding sets of a and c are included in each other. □

Lemma 5.2. *The strict domination relation is well-founded.*

Well-founded means that in any set, there is a minimal element. In our case that means that there is a line that doesn't strictly dominate any other line. This property is used later to perform induction on lines.

Proof. Define the weight of a line as the sum of its sets' cardinalities. Obviously no line can have a negative weight. If a line a strictly dominates line b, then the weight of b is less than the weight of a, since all of b's sets are subsets of a's and one is a strict subset.

A suitable minimal line for any set is the line with the lowest weight. If there was a line strictly dominated by the lowest-weight line, that line would have even lower weight, which is a contradiction. □

5.3 Building lines

A *singleton line* is a line that generates just one configuration. It is the configuration with each symbol wrapped in a set.

Lemma 5.3 (Line splitting)**.** *For any non-singleton line x there exist two lines strictly dominated by x that can be combined into x.*

Proof. Since x is not a singleton line, one of its sets, $S = \{a, b, \dots\}$, must contain multiple elements. Make two lines: one where S is replaced with $S \setminus a$ and another where S is replaced with $S \setminus b$. Their union is S, so x can be built out of them and x strictly dominates them, as they contain strictly less symbols. □

Theorem 5.2. *Any line can be built by combining singleton lines that it dominates.*

Proof. One can perform well-founded induction on lines because the strict domination relation is well-founded. Thus it suffices to show that if all lines strictly dominated by x can be built from dominated singleton lines, x can be, too.

If x is not a singleton line, we can use the line splitting lemma (Lemma 5.3) to show that it can be built out of two lines that it strictly dominates. The induction hypothesis tells us that those lines in turn can be built from dominated singleton lines. □

Lemma 5.4 (Bigger is better)**.** *If lines a and b can be combined into c then a' that dominates a and b' that dominates b can be combined into c' that dominates c.*

It is clear that if a' and b' are combined in the same fashion as a and b, the result cannot contain less symbols.

Theorem 5.3 (Combination is complete)**.** *A line dominating any valid line x can be built by combining input lines.*

Proof. According to Theorem 5.2 x can be built from singleton lines that it dominates. Those singleton lines are valid as well, since dominated lines generate strictly less configurations. Being valid singletons, they generate one configuration each, which is contained in some input line.

If one combines those input lines instead, the result is a line dominating x according to Lemma 5.4. □

Corollary 5.3.1. *The maximal lines can be built by combining input lines.*

Proof. A line is maximal if no valid line strictly dominates it. Thus a valid line dominating a maximal line must be equal to it. Lines obtained by combining input lines are valid; Theorem 5.1 shows that combining does not enable new configurations. □

5.4 Efficiently finding the maximal lines

Even if combining lines eventually produces the maximal lines, it may not be an efficient way of finding them. Building a maximal line could require hundreds of combinations!

Define a *missing line* as a valid line dominated by none of the input lines.

Lemma 5.5. *A line that dominates a missing line is also missing.*

Proof. Let a and b be any lines such that b dominates a and a is missing. Suppose that b is not missing. Then there is an input line i that dominates b. Because the domination relation is transitive (Lemma 5.1) i dominates a as well, so a is not missing, which is a contradiction. □

Theorem 5.4. *If there is a missing line, then some missing line can be obtained by combining two input lines.*

In other words, we can make progress by simply trying all combinations of two lines.

Proof. According to Theorem 5.3, there is some way of combining input lines that produces the missing line. Since they are combinations of the input lines, all lines leading up to the missing line are valid due to Theorem 5.1.

By definition a line that is valid but not missing is dominated by some input line. Thus all of them are either missing or dominated by an input line.

Suppose that the lines that combine into the missing line are dominated by input lines. Combining those input lines in the right way yields a line that dominates the missing line according to Lemma 5.4. Lemma 5.5 tells us that bigger line is missing as well.

If one (or both) of the lines are missing, recurse into the missing line. There will be a pair of non-missing lines eventually, as the combination starts with input lines. □

Theorem 5.4 gives an efficient process for checking maximality: if all ways of combining two lines result in lines dominated by some existing line, then the current set of lines is maximal.

A trivial extension yields an algorithm for finding the maximal form: when a line that is not dominated by any existing line is found, add it to the set of lines and try combining it with all the other lines. This process is bound to terminate, as strict domination is a well-founded relation. When it does, filter out all dominated lines and only the maximal lines are left.

Before explaining the proof in Coq, I will explain the theoretical underpinnings of it and other type theory based proof systems.

6 Type theory as mathematical foundation

The Curry-Howard correspondence states that a program is a proof of a theorem corresponding to its type. Thanks to this idea, we can use a model of computation as the foundations of mathematics instead of the conventional Zermelo-Fraenkel set theory axioms.

The book Homotopy Type Theory [33] covers most of the information in this section among other things.

6.1 How to prove with code

The notation $e : T$ is read as expression e has type T. This is analogous to set membership in set theory if T is seen as the set of all values that e could have. Types are not sets and do not always behave in the same way but some operations look very similar for both. Notably, types don't support union and intersection like sets do.

Types that are checked at runtime, found for example in Python are not the same thing. In Python, the same expression can have a different type on different runs. I am only talking about statically typed languages, so every expression has just one type, which can be computed without running the program.

Constructing a value of type T proves that something of type T exists. However, in most programming languages it is possible to write a function that does not construct a value of type T even though that is its return type. Apart from broken type systems, this happens because the languages allow programs to crash or loop. Section 7.6 explains how Coq prevents that.

We can now prove that integers exist by writing $1 : \mathit{Integer}$, which is not very exciting. We can also encode propositional logic (see Listing 1) and integers. Some programming languages' type system is even Turing-complete but as far as I know, no type system is seriously used as a programming language. It would be much more useful to write theorems about the outputs of functions instead of some ugly type-level constructs. This is where dependent types come in.

```
let a_or_b: Result<A, B>;
let a_and_b: (A, B);
let not_a: &dyn Fn(A) -> Void;
enum Void {}
```

Listing 1: Propositional logic in Rust types. Initializing these variables in safe Rust would prove what the name implies if Rust was total (see Section 7.6).

A dependently typed programming language can express types that depend on values as opposed to simply typed languages, where types and values are entirely separate. For example, with dependent types, one is able to write a type $1 + 1 = 2$ where 1, 2 and + are the same constructs that one would use in a program that performs calculations. Even quantifiers are implemented with dependent types in Coq.

Before going into dependent types it more detail, I will briefly discuss the algebraic notation that is often used when discussing types.

6.2 Type algebra

The unit type 1 is a type with only one inhabitant, commonly denoted as the empty tuple `()`. The type 0 is a type that no value has. It should not be confused with **void** in C, which is actually 1. If a function's return type is 0, the function must never return, as it is impossible to construct a value of type 0.

Values of a product type $A \times B$ consist of an $a : A$ followed by a $b : B$. The Cartesian product is the corresponding set operation. Many programming languages have a tuple type, which is a pure product type. Record or struct types are common named variants of product types. In most cases, they could be replaced with tuples; the names are only for making them more readable.

Values of a sum type $A + B$ are either of type A or of type B. The corresponding set operation is the disjoint union. A disjoint union is a union where the elements are augmented with an index indicating the set from which set they originate. In programming, this is known as tagged union. Enumerations are the simplest example; Booleans could be written $1 + 1$, though it is more concise to call them 2. A tagged union with zero variants cannot be constructed so it is 0. Optionals found in Rust or Scala among others, are of the form $T + 1$. Structurally typed unions, an intermediate between C's wildly unsafe unions and tagged unions is found in gradually typed languages like TypeScript and in the joke language IntercalScript [17].

Knowing this much type algebra is sufficient to understand why the dependent types presented next are called Π and Σ-types but I will go on to show that types equipped with the previously defined sum and product form a semiring just to highlight that there truly is an algebra of types.

6.2.1 Aside: More type algebra

I have many times found myself wondering if I should write a data structure $A \times (B + C)$ or $(A \times B) + (A \times C)$ when it is not entirely clear if A really represents the same thing in both cases. I am not aware of any programming language where such *isomorphic types* can be used interchangeably. Incidentally, in Coq $((1, 2), 3)$ is equal to $(1, 2, 3)$ but $(1, (2, 3))$ has a different type.

Theorem 6.1. *The operations $+$ and $\times$ form a semiring on the isomorphism classes of types.*

Proof. It is clear that $+$ is associative and commutative. Its neutral element is 0; a variant holding zero will never be constructed, so an added 0 does nothing.

We can translate between $((a, b), c)$ and $(a, (b, c))$, so $\times$ is associative. The neutral element of $\times$ is 1, as it doesn't add any information, like struct padding in C.

Distributivity: $A \times (B + C) \cong (A \times B) + (A \times C)$. As discussed earlier, 0 is impossible to construct, so a tuple containing it is, too: $A \times 0 = 0$. □

Finding an isomorphism, a one-to-one mapping, is equivalent to showing that two sets have the same cardinality, so finite types can be identified by natural numbers. We already saw the types 0, 1 and 2. For finite types, one could just have stated that $|A + B| = |A| + |B|$ and $|A \times B| = |A||B|$ and used the fact that natural numbers form a semiring. The problem with non-finite types is that depending on the type system, they may not correspond to ordinals or any other well-known kind of integers.

From the cardinality point of view, it is easy to see that the arrow $\to$ in function types corresponds to exponentiation; a function is just a table of outputs with an entry for each input.

$$|A \to B| = |B|^{|A|}$$

A table of things of type T is just T raised to some power.

Inductive types have interesting equations, for example

$$\text{List} = 1 + (A \times A) + (A \times A \times A) + \ldots$$

is a solution to

$$\text{List} = (A \times \text{List}) + 1,$$

which is the typical definition of a list in functional programming languages. Conor McBride [23] has even shown that the partial derivative of a type is a data structure representing an in-progress traversal of it.

6.3 Examples of dependent types

6.3.1 Dependent functions

A Π-type is a type that depends on the value passed in. For example, ordinarily it would be impossible to write a function that sums all entries in a tuple because tuples of different lengths have different types. Using a Π-type, the function could have the type $\Pi_{n:\mathbb{N}}\ \mathbb{N}^n \to \mathbb{N}$.

In dependently typed programming languages, Π-types typically look like ordinary functions. For example, the previous example can be written in Coq as

```
Fixpoint ntuple n T : Type :=
  match n with
  | 0 => T
  | S n' => ntuple n' T * T
  end.

Notation "T ^ n" := (ntuple n T).

Fixpoint sumt (n : nat) (nats : nat^n) :=
  match n return nat^n -> nat with
  | 0 => fun x => x
  | S n => fun a => let (r, x) := a in sumt n r + x
  end nats.
```

The universal quantifier `forall x : T, P x` is another way to write a Π-type in Coq.

Why is it called a Π-type? A type $\Pi_{e:T}\ D(e)$ could be seen as the image $D[T]$, which contains the value of D for all values that e can have. That image is a product type, which is denoted with Π just like products on numbers.

6.3.2 Dependent pairs

If the naming of Σ-types follows the same pattern as the naming of Π-types, then $E = \Sigma_{e:T}\ D(e)$ is the sum of the image $D[T]$. And it is! A value of type E contains just one of the values that $D(e)$ can have. And we know the value of e, too, because $+$ is a disjoint union.

In dependently typed programming languages, Σ-types often look like ordinary data structures. One could add a data structure that stores a tuple of arbitrary length to the previous example:

```
Inductive tlist T := make_tlist (n : nat) (l : T^n).
```

Explicit Σ-types are called dependent pairs by programmers. The Coq standard library defines syntax for dependent pairs that allows writing `tlist` as

```
Definition tlist2 T := { n & T^n }.
```

Viewed as a proposition, $\Sigma_{x:T}\ P(x)$ corresponds to $\exists x : P(x)$.

6.3.3 Other exotic types

Sigma- and Π-types are not the only kinds of new types that can exist in dependent type theories [19]. For instance Rathjen [16] has shown that extending Martin-Löf type theory with well-founded tree types makes it a significantly stronger proof system.

Coq's `Inductive` definitions can be used to define many kinds of dependent types but not all of them; for example higher inductive types are not supported.

6.4 Universes

A universe hierarchy is a feature of many dependently typed languages that the user mostly does not encounter but that is required for consistency as a proof system. It prevents making a paradoxical type P such that $P : P$. In a language with polymorphism, such a type can be used to prove anything via Girard's paradox.

In Coq, the type of a simple type is Set, the type of set is Type0, the type of Type0 is Type1 and the type of that is Type2 etc. Types have subtyping: any type may be used as if its universe index was greater [8].

Impredicativity tends to cause inconsistency in logical systems. Coquand [12] reports on one such paradox he has discovered. The universe hierarchy forces predicativity, meaning that an object can not take itself as an argument. Such an object would have to have to live in a universe U such that $U < U$.

Certified Programming with Dependent Types [8] gives an example of a universe error (shown below) but it has to be slightly modified, as it compiles and runs without errors on Coq 8.15.

```
Definition id {T: Type} (x: T) := x.
Check id id.
```

The code typechecks because Coq's type inference has become good enough to figure out that it can choose `T := U -> U` in the first `id` and `T := U` in the second.

To get the error that the example tries to produce, we need to disallow inferring the implicit argument.

```
Check id (@id).
```

produces the expected error:

```
The term "@id" has type "forall T : Type, T -> T"
while it is expected to have type "?T"
(unable to find a well-typed instantiation for "?T":
cannot ensure that
"Type@{id.u0+1}" is a subtype of "Type@{id.u0}").
```

Especially with the improvements in inference, typical users rarely encounter universes. There is already experimental support for universe polymorphism [1] which completely eliminates the problem in the above example by allowing use of the same definition with different universe levels.

7 Mechanized proof

I have formalized the proof presented in Section 5, written an implementation of maximality checking in Coq (depicted in Listing 3) and proved that implementation's correctness. The statement of Theorem 5.4 in Coq is shown in Listing 2. The proof's source code [27] is publicly available as text and in a format suitable for reading in the browser. More on that in Section 7.1.

```
Theorem combining_two_suffices (x : line):
  missing x -> nonzero x -> valid x ->
  ∃ a b c, a \in input ∧ b \in input ∧ combination_of a b c ∧ nonzero c ∧ missing c.
```

Listing 2: The biggest theorem in the Coq proof that shows that one can find missing lines by combining two lines.

This section is written for a reader that has some familiarity with statically typed functional programming but has not used Coq. I would encourage readers that have already written some mechanized proofs of their own to study the proof's source code along with the natural language proof (Section 5) that closely follows it.

```
Definition cannot_find_more :=
  [forall a in input, [forall b in input,
    [forall c in all_combinations a b, ~~ (missing c && nonzero c)]]].

Definition none_dominated :=
  [forall a in input, forall b in input, ~~ (a ⊂ b)].

Definition is_maximal_form :=
  cannot_find_more && none_dominated.

Theorem is_maximal_form_works :
  is_maximal_form <->
  (∀ x : line, nonzero x ->
    [exists y in input, perm_eq x y] <-> maximal x).
```

Listing 3: The definition of the verified maximality checking function and the theorem that states that `is_maximal_form` returns true only iff the input consists of the maximal lines. That is, every line in the input is maximal and every maximal line is in the input.

7.1 How to read Coq

Coq very closely resembles OCaml, the language it is implemented in. The main syntactic difference is that top-level definitions start with a capital keyword and end with a full stop.

The majority of those statements are proofs, which may be started with **`Lemma`** or **`Theorem`**. They can be used interchangeably. I have used **`Theorem`** to mark the

most important ones, which are also called theorems in the natural language proof (Section 5). Between the keyword and the first stop is what is to be proved. The the proof follows, terminated by **Qed** or **Defined**. Proofs terminated by the latter are allowed to be executed as programs.

Printed on paper, the proofs are typically completely incomprehensible. Try reading Listing 4 for example. The reason is that they manipulate the state of the proof but that state is invisible in the source code. This is the complete opposite of paper proofs, which typically report the state after every transformation rather than how they got there.

```
Lemma permutation_contains :
  ∀ a (s s' : line), perm_eq s s' -> a ∈ s = a ∈ s'.
Proof.
  move=> c.
  apply: perm_rew_helper => s s' pe.
  rewrite /contains/contains_unpermuted.
  move=> /existsP[cs /andP[pe_cs q]].
  move: pe; rewrite perm_sym; move=> /tuple_permP [mapping smap].
  fill_ex [tuple tnth cs (mapping i) | i < Δ].
  apply/andP; split.
    apply: perm_trans; last by apply: pe_cs.
    by apply: any_perm.
  move: q => /all2_tnthP q.
  rewrite smap; apply/all2_tnthP; move=> i.
  rewrite !tnth_map.
  by apply: q.
Qed.
```

Listing 4: A sample proof. It should be clear what it proves but the proof itself is unreadable.

To allow curious readers to see everything without having to install anything, I have generated a web page [1] with Alectryon [26], which allows browsing the intermediate steps.

Proofs look different from the rest of the code because Coq is a combination of the pure functional programming language Gallina and Ltac, an untyped logic programming language used to generate Gallina. I will not show any more proofs due to the readability issue. However, Section 7.7 explains how Ltac works.

7.2 Notations

Coq source code can look more exotic than OCaml because it is very easy to extend the syntax of Gallina.

My code uses the syntax defined in Mathematical components, for instance `n.-tuple T` is a length n array with elements of type T and `'I_n` is the type of integers from zero to $n-1$.

[1] https://joonazan.github.io/line-combination-proofs/

I define some binary operators for convenience: $\subseteq$ to denote domination, $\subset$ for strict domination and $\in$ to denote that a line contains a coloring.

```
Notation "a ⊆ b" := (dominates b a) (at level 50, no associativity).
```

Notations are not limited to binary operations or fixed size syntax. For example the list literal `[a; b; c]` is defined as a notation in the standard library.

7.3 Variables

Variables like

```
Variable alphabet_size: nat.
```

can be used in function definitions within the current section. If a function that uses them, even indirectly, is called from another section, the variables need to be given as arguments.

For example, my definition of the type `color`

```
Definition color := 'I_alphabet_size.
```

uses `alphabet_size`. My definition of configurations

```
Definition configuration := Δ.-tuple color.
```

uses `color`. Now configuration takes `alphabet_size` (and `delta_minus_one`) as arguments, even though the definition has no parameters.

Variables are very convenient; without them I would have to explicitly make almost every lemma depend on the alphabet size and graph degree.

I use the variable `delta_minus_one` instead of Δ to ensure that Δ is positive. That way I do not need to deal with the uninteresting case where vertices are not connected at all. I could instead add `Variable nonzero_delta := Δ <> 0` but then I would have to explicitly use it every time to eliminate the zero case.

Encoding a natural number that is at least k as $n + k$ composes very well. For example calling a function that needs something of size $m + 1$ with something of size $n + 2$ works. Both Coq's default nats and ssreflect's nats are Peano numbers, so in the aforementioned scenario the type inference engine has to unify `S m` with `S (S n)`, which results in a substitution of `m` by `S n`.

7.4 Representation of lines

I chose to represent lines and configurations as tuples, so their lack of order is simulated by operating on some permutation instead of the actual line. This choice of representation is not ideal as discussed in Section 9.1.

```
Definition nline n := n.-tuple {set color}.
Notation "n .-line" := (nline n) (at level 30, no associativity).
Definition line := Δ.-line.
```

Both containing a configuration and beign a combination of two lines is defined as an auxiliary procedure operating on two ordered lines and a predicate that holds if there is some permutation of the lines that satisfies the auxiliary procedure.

```
Definition contains_unpermuted (cl: configuration) (l: line) : bool :=
  all2 (λ c (s : {set color}), c \in s) cl l.

Definition contains (cl : configuration) (line : line) : bool :=
  [exists (p : configuration | perm_eq p cl), contains_unpermuted p line].

Notation "a ∈ L" := (contains a L) (at level 50, no associativity).
```

The choice of permuting the configuration in `contains` is arbitrary. Proving `perm_eq s s' -> a ∈ s = a ∈ s'` would be a lot easier if the line was permuted, so it could be argued that a symmetric definition where a permutation of the configuration is compared against a permutation of the line would be best.

```
Definition combine m (n := m.+1) (a : n.-line) (b : n.-line) : n.-line :=
  [tuple of (thead a) :|: (thead b)
  :: map (λ ab, ab.1 :&: ab.2) (zip (behead a) (behead b))
  ].

Definition combination_of (a b c : line) :=
  ∃ (a' b' c' : line),
    perm_eq a' a ∧ perm_eq b' b ∧ perm_eq c' c ∧ combine a' b' = c'.
```

Because I never perform case analysis on `combination_of`, it is defined as a proposition, whereas contains is defined as a decision procedure.

7.4.1 Zero lines

The paper proof never deals with lines that contain an empty set because those are not lines. The line data structure defined here allows those invalid lines with zero configurations. Most proofs hold for them as well some of the proofs leading up to the conclusion only accept proper lines by requiring them to satisfy

```
Definition nonzero := all (λ x : {set color}, #|x| != 0).
```

7.5 Inductive definitions

An **`Inductive`** definition in Coq is very similar to what is known as a generalized algebraic data type (GADT) in Haskell.

Type constructors in general can be thought of as opaque functions that can be performed in reverse. GADTs make this explicit, as they are defined by writing the type of each type constructor. Like most things in Coq, the constructors in inductive definitions can be dependently typed.

Depicted below is the only inductive definition in my code. They usually define data structure but mathematical components already contains the data structures that I need, so this one is used as a specification. It is a proof that a line can be

obtained by repeatedly combining input lines. There are two cases: either the line is in the input, or the line can be constructed from two other lines that can be constructed out of input lines.

```
Inductive iterated_combination : line -> Prop :=
    present : ∀ a, a \in input -> iterated_combination a
  | combined : ∀ (a b c : line),
      iterated_combination a -> iterated_combination b ->
      combination_of a b c -> nonzero c -> iterated_combination c.
```

Data structures like this are just like Prolog specifications. The same could be written in Prolog as follows.

```
iterated_combination(A) :- in_input(A).
iterated_combination(C) :-
  combination_of(A, B, C),
  iterated_combination(A),
  iterated_combination(B).
```

But Prolog has a fixed evaluation strategy. In Coq, the evaluation strategy is controlled by writing tactics (see Section 7.7).

These kinds of specifications are generally very easy to read but in this one it is not clear why `c` needs to be `nonzero`. But that doesn't matter because this one is only used in an intermediate step, rather than the final theorem's specification.

7.6 Totality

Every function in Gallina is *total*, which means that all programs eventually complete without crashing if executed on correct hardware with a sufficient amount of memory.

Totality is necessary because it would be easy to prove anything with a non-terminating function, as the return type of a function that never returns can be anything.

```
Fixpoint make_false n : False := make_false (n+1).
```

Fortunately attempting to define the above function produces the error 'Recursive definition of make_false is ill-formed.' Coq requires that recursive functions have at least one *decreasing argument* [1]. In recursive calls, the decreasing argument must be *structurally* smaller; It must be only part of what was passed in. For example, a function that takes a list could call itself with just the tail of the list.

It can be shown that Gallina is not Turing-complete because all Gallina programs eventually halt. It is well known that checking if an arbitrary program halts is undecidable. But the halting problem is trivial for all Gallina programs, so Gallina can express only a subset of all programs.

In practice the limitation is rarely detrimental. Totality prevents writing a function that produces the Collatz sequence starting from an arbitrary integer because nobody has been able to prove that that sequence is always finite. One can still compute the first million numbers, which is the same thing for all practical purposes.

Deliberate infinite loops are a more common occurrence than problems that push the limits of mathematics. For example a web server is usually supposed to serve until it is stopped. However, serving one request must complete, so it suffices to wrap a Coq program in an infinite loop. In general, one can write a Coq program that computes a new state based on an event and feed it one event and the current state in an infinite loop.

Like recursive functions, inductive data structures are also limited to avoid infinite loops. They have to satisfy the positivity condition, which I will not explain in detail, as knowing it isn't necessary to write this the proof discussed in this thesis. However, I will give an example of a data structure that would lead to nontermination if allowed.

Suppose there was a data structure

```
Inductive Wrapper T :=
  Wrap : (Wrapper T -> T) -> Wrapper T.
```

that wraps a function that takes a `Wrapper T` and returns something of type `T`. (That data structure is disallowed by Coq's positivity check.) Now we can write a completely safe recursion-free function

```
Definition unwrap {T} (w : Wrapper T) : T :=
  match w with Wrap f => f w end.
```

that takes a Wrapper and calls the function inside it with itself.

The previous function is actually just a helper for

```
Definition materialize T : T :=
  unwrap (Wrap unwrap).
```

which returns something of type `T` for any `T`. Which is very bad because it can create a proof for any statement, even false statements. It does that by calling `unwrap` with `Wrap unwrap` in an infinite loop.

7.6.1 Non-structural induction

Not all total functions have a decreasing argument. My proof of

```
Theorem combination_is_complete :
  ∀ line, valid line -> nonzero line -> ∃ line',
    iterated_combination line' ∧ line ⊆ line'.
```

is a recursive argument that repeatedly applies to lines strictly dominated by the current line until eventually reaching a singleton line (see Section 5.3). A strictly dominated line is not structurally smaller, but it is still relatively easy to see that a chain of strictly dominated lines cannot go on forever. In other words, it is a well-founded relation.

Coq's standard library contains proofs for utilizing well-foundedness. They don't extend Coq's totality checking, they convert well-founded induction to structural induction.

I use a weight function to map lines to natural numbers just like in Proof 5.2.

```
Definition weight (l : seq {set color}) := \sum_(i <- l) #|i|.
```

It is easy to prove `well_founded strictly_dominated` using the helper lemma

```
Lemma strictly_dominated_wf' : ∀ len (a : line),
  weight a < len -> Acc strictly_dominated a.
```

since `well_founded` is defined as `forall a, Acc R a`, meaning that every inhabitant of a type is accessible via the relation R. Accessibility is enforced by the `Acc` type's sole constructor

```
Acc_intro : (forall y:A, R y x -> Acc y) -> Acc x.
```

which requires that all smaller elements are also accessible. (The minimal elements are accessible because there are no smaller elements with respect to them.)

The helper lemma is proved by performing induction on weight and applying the lemma

```
Lemma strictly_dominates_weight (a b : line) :
  a ⊂ b -> weight a < weight b.
```

relating weight to strict domination. With well-foundedness proved, we can use `well_founded_induction` from the standard library in `combination_is_complete`.

7.7 Ltac

Ltac is the tactics language of Coq. It is mostly used to generate proofs. Sometimes it is used to derive a function whose signature forces the definition to be correct, for example decidable equality, a function that returns a proof that its arguments are equal or a proof that they are not equal. If a function is tricky to define because it requires proofs it is sometimes defined using the **`refine`**-tactic, which takes Gallina code with blanks and allows filling those blanks using Ltac.

Ltac is not meant for generating normal code or specifications; its library focuses on generating any code that fits a type and it is less trustworthy than the core of Coq.

Code generation is a game of filling in blanks. At the start, there is one blank for the whole function body. It is replaced with some code that in turn contains one or more blanks. When all the blanks have been filled, the function is complete.

The blanks are just like typed holes in Haskell; they show the user the expected type and the typing environment i.e. the variables that are accessible from the hole. Coq calls the type of the hole the goal and the environment the set of hypotheses. Figure 8 shows how they look to a user.

Note that the hypotheses only show the local environment. Things defined outside the current function are also in the environment but they are not shown, as there can be thousands of them. There are commands like **`Search`** for quickly finding a helpful lemma out of the full environment.

The basic building blocks of Ltac are tactics written as plugins in OCaml. One of these tactics can be invoked by writing its name and possibly some arguments followed by a stop. For example, **`split`**. breaks goals like `A /\ B` into multiple goals.

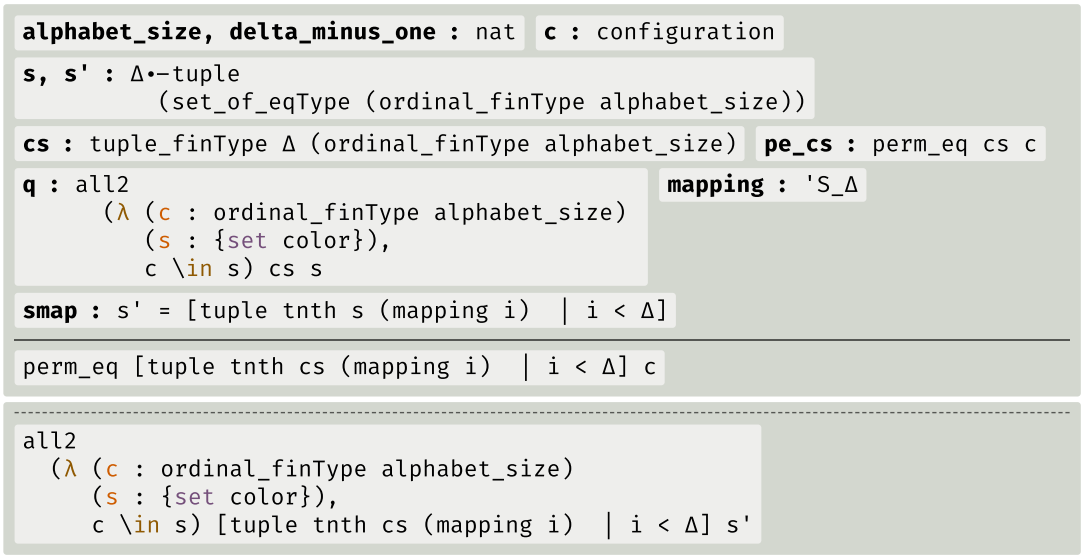


Figure 8: The proof view. At the top are the hypotheses, below the solid line is the goal and below the dashed line is a currently unfocused goal.

Multiple tactics can be chained together with a semicolon. If a tactic results in multiple new goals, the second tactic is applied to each one of them. For example, if `A` and `B` from the previous paragraph are easy to prove, `split; auto.` could completely prove it.

Useful combinations of tactics can be given their own name. For example, I have written

```
Ltac split_and := repeat (split || (apply/andP; split); try done).
```

which splits everything it can and tries to convert Boolean ”and” to propositional ”and” if it can not split and automatically proves all trivial goals using the **done** tactic from ssreflect.

Ltac can even select a tactic by pattern matching on the goal and hypotheses. I do not use it in this development but it can be very useful, especially when automating large parts of a development. A significant limitation of the technique is that the match is purely syntactic; unlike equality in Coq, pattern matching in Ltac does not consider two terms that evalute to the same normal form equal.

Mathematical components defines a lot of tactics and tactic notations (same as a notation but for Ltac) that I use, so my proofs do not look like standard Coq. The reason for the abundance of notations is that the library offers highly compact notation for writing precise tactics rather than fixing the lack of precision with powerful automation.

7.8 Mathematical components

Dealing with permutations seemed like something that could benefit from existing theories, so I switched to the Mathematical components (math-comp) library at that

point in the development. It was originally created as part of a formal proof of the Four color theorem. It contains theorems about abstract algebra and common data structures.

Mathematical components is built on the small-scale reflection (ssreflect) library, which has a strong opinion on how Coq proofs should be written and provides utilities for writing in that particular style. I have adopted this style in my proof.

7.8.1 Reflection views

An important idea in ssreflect is that every proposition is written as a Boolean expression and converted into the most suitable form via a reflection view when necessary [21].

The Boolean form is excellent for case analysis and proofs via computation. Case analysis on propositional forms is not possible without adding the law of excluded middle $\forall a : a \vee \neg a$ as an axiom. It is common to avoid axioms if possible in Coq because not all axioms are compatible with each other.

Suppose we would like to separately prove the case where a is equal to b and the case where they differ. Via small-scale reflection we can view their equality as a Boolean and perform case analysis on it. In conventional Coq, one would have to write a function for the type of a and b that either returns that its arguments are equal or that they are not. For equality, the standard library defines a tactic `decide equality` that fills in the body of such a function but that only works for equality. Ssreflect provides reflection conversions between Boolean and propositional forms of equality, implication, negation and quantifiers among others.

The propositional forms are useful because unlike the Boolean forms, they are data structures that can be operated on. An assumption that is a conjunction can be broken down into two assumptions. Case analysis can be performed on a disjunction. Propositional equalities can be used to rewrite.

7.8.2 The ssreflect proof language

Ssreflect also focuses on maintainability of proofs. A small change somewhere could invalidate a proof. Because many tactics never produce an error, code that previously completed a subgoal may leave that goal incomplete, causing weird errors in the tactics that were meant to be applied to the next goal. To prevent this, ssreflect advocates the **by** tactical, which produces an error if the tactic following it does not discharge a goal.

Another issue that ssreflect attempts to remedy is that most of the standard tactics in Coq create assumptions with automatically generated names. The names can be confusing and the only way to find out where some assumption `H1` is coming from is to step backwards in the proof until it dissapears. Because names are assigned sequentially, changing the beginning of a proof can change the name of every assumption, leading to a lot of tedious renaming.

Ssreflect's solution to naming is to add new assumptions to the goal so they do not immediately need a name. After adding an assumption `X` to a goal `G`, the new goal is `X -> G`. There are many bookkeeping tactics to rearrange the chain of

implications like the stack in a language like FORTH.To allow the user to completely avoid automatic naming, ssreflect contains its own set of tactics that replaces almost all standard tactics.

When shuffling assumptions around isn't enough, they can be removed from the goal and named with the `=>` tactical, which can be appended to any tactic. For example, to name two assumptions produced by the tactic `tac`, one could write `tac => a b`. There is in fact a pretty rich language that one can use after the arrow. Most importantly, breaking products into their parts is done with a pair of square brackets and applying a function or reflection view is done with a slash. For example `/andP[a b]` takes the first assumption in the stack, turns it from a Boolean and to a propositional and, splits it into two assumptions and names them `a` and `b`.

Instead of manual naming, Chlipala [8] prefers to write very short, almost completely automated proofs. They are maintainable because they do not refer to named assumptions, instead defining automated procedures for processing certain shapes of goals or assumptions. My attempts at that level of automation have not been very successful but I can see that my proof could be significantly shorter if some of the more trivial work was automated. Proofs with both heavy automation and cleaner presentation in the style of ssreflect could even be somewhat readable.

A significant downside of Mathematical components is that it defines its own version of everything. Even the `nat` in ssreflect is incompatible with the standard library's `nat`. Because of this standard **`lia`** tactic, which automatically solves linear integer arithmetic (equations involving comparison, addition and multiplication with constants) can only be used after converting every expression to the equivalent on standard nats. I solved this problem by writing a tactic called liafy that converts a subset of mathematical operations.

```
Ltac liafy :=
  rewrite -?(rwP leP) -?(rwP ltP) -?(rwP negP) -?(rwP eqP) -?plusE.
Ltac sslia := liafy; lia.
```

Thanks to Mathematical components I did not have to write any data structures myself, which makes me think that it is well suited for people who know abstract algebra very well but know little about programming. Writing the basic operations on a data structure in Coq can be very unintuitive especially because the convoy pattern [8] is often necessary.

For example, when pattern matching on an array's length, the code in the match arm corresponding to zero still does not know that the array is empty. It seems that there is no way around it other than proof mode but actually one can very tediously annotate the pattern match, which makes it work in the desired way.

Pattern matching in many other languages is much easier to use but they typically rely on axiom K for pattern matching, limiting the settings the language can be used in; for example, Homotopy type theory's Univalence axiom is incompatible with axiom K. Cockx et al. [9] developed and formally proved a version of dependent pattern matching without K in 2014 that is available in Agda today.

8 Performance

Maximization is the most expensive part of the original Round Eliminator's algorithm. Maximization via line combination is orders of magnitude faster in practice and is resistant to some pathological inputs but it remains the most computationally expensive part of round elimination.

Evaluating the performance of maximization is difficult, as its output can be exponentially larger than its input. However, there is no known algorithm for checking maximality that is significantly better than performing maximization. Therefore I will compare different algorithms' performance on the decision problem of finding out whether a set of lines is maximal or not.

8.1 Time complexity of the best previously known algorithm

The maximization algorithm previously used in Round Eliminator relies on a procedure that transforms a maximal form into a maximal form with one configuration removed. If every configuration was valid, the maximal form would be a single line that generates every configuration, so that line is the starting point. Then each invalid configuration is removed one by one.

The algorithm sees all permutations of a line as separate lines and deduplicates only at the end. I will refer to them as lines and configurations in this description even though I mean their ordered counterparts.

To remove an invalid configuration from a line, the line is split into Δ different versions that cannot produce the invalid configuration because one of the symbols in the configuration has been removed from each.

The algorithm computes the set of invalid configurations by filtering all $|\Sigma|^\Delta$ possible configurations. It produces $\Delta!$ permutations of the output lines before the removal of permutations.

After removing each configuration, dominated lines are discarded with an $O(\Delta n^2)$ pass.

It is unclear how exactly the size of the maximal form develops, so the average number of lines is unknown. Assuming that the number of lines stays within a constant factor of the number of lines at the end, the time complexity of checking maximality is $O(|\Sigma|^\Delta (n\Delta!)^2) = O(|\Sigma|^\Delta n^2 (\Delta!)^2)$.

There is an adversarial input that defeats this algorithm. Just concatenate multiple copies of an input, renamed so that each has its own disjoint subset of the alphabet. Then the number of valid lines grows linearly with the size of the input but the number of possible configurations grows to the power of the degree.

8.2 Time complexity of line combination

To check maximality with the new algorithm, one has to combine every pair of lines in every possible way and compare the results to the original lines.

The lines can be combined in $n^2 \Delta \Delta!$ different ways, n^2 ways to choose lines, $\Delta!$ ways to match them up and Δ ways to choose which pair to take the union of.

However, most of those ways result in lines that can be discarded because they contain an empty set or because they are dominated by the original lines or some other combination.

Computing the combinations is feasible even for large Δ when the lines contain many duplicated sets because in that case the order of the duplicated elements has no effect on the outcome. This seems to happen often in practice. However, it is an open question if it is possible to design an algoritm that does not have a factorial worst case.

Comparing two lines can be done by building a bipartite graph where each bipartition represents one of the lines and connecting one side's sets to every set on the other side that is included in them. If there is a perfect matching, one line dominates the other. Bipartite perfect matching can be solved in $O(\Delta^{2.5})$ time via the Hopcroft-Karp algorithm. In practice it seems that a backtracking search can be faster but that could be asymptotically worse.

The resulting time complexity is $O(n^2\Delta\Delta!n\Delta^{2.5}) = O(n^3\Delta^{3.5}\Delta!)$. Notably, the size of the alphabet has no effect on running time anymore.

In practical implementations there are tricks for quickly discarding most results of line combination, so the power of n is closer to 2 than 3. But I have not proved that.

Another benefit of the new algorithm is that checking maximality with it requires only $O(1)$ extra space, while the old one requires $O(|\Sigma|^\Delta + n\Delta!)$. Practical implementations of the new algorithm use some extra memory for a cache of previously discarded lines.

8.3 In practice

I and Dennis Olivetti both implemented the new algorithm in the Rust programming language. Both are publicly available at https://github.com/joonazan/maximizer and https://github.com/olidennis/round-eliminator (branch round-eliminator-2) respectively.

Both implementations compare new lines against other lines originating from the same pair of lines before comparing against the rest. This is worthwhile based on the assumption that many of the new lines are just inferior versions of the best lines of that batch.

Olivetti showed me that maintaining a set of dominated lines and checking new lines against that set before doing any comparisons improves performance by an order of magnitude in practice, which means that at least in the inputs tested, tens of copies of some useless lines are created.

A problem that took Olivetti's implementation of the old algorithm several minutes was solved in a hundredth of a second by my implementation of the new algorithm. Olivetti managed to run the new algorithm on problems with $\Delta > 10$ by optimizing for the case when lines contain the same set multiple times. Those problems were previously infeasible.

I would have liked to measure the performance of the function `is_maximal_form` (shown in Listing 3) but it turns out that even on the smallest possible input it

requires a finite but large amount of memory. The next section explains how that shortcoming could be rectified.

9 Discussion

My goal was to build a provably correct version of round elimination while also improving its performance. I chose to focus solely on maximization, as it is the only nontrivial part of round elimination.

The line combination algorithm for maximization delivers on the performance goal and is already in use but the machine-verified procedure for checking maximality needs more work to be of practical use. This section outlines how one could make the maximality checking program practical and how the proof could be extended to cover round elimination or even whole proofs of time complexity upper or lower bounds.

9.1 Impractical data structures

The formalization of round elimination uses many data structures and proofs provided by the Mathematical Components library. However, that library is made with pure mathematics in mind; functions using the data structures are not meant to be executed.

One example is the definition of `n.-tuple`, a fixed size list. It is simply a wrapper around a list and a proof that the list's length is `n`. As the tuple goes through various transformations, the proof grows longer, as it is always replaced with a new proof that references the old one.

I have observed that the proof's memory use grows much faster than linearly with the length of the tuple. I suspect that this is due to some proof that iterates through all permutations of the tuple. In any case, it is clear that actually executing the proof code is not viable.

The most common solution to this kind of problem is proof irrelevance. The idea is that theorems can be types that never have more than one inhabitant. That forces proofs be compile-time only: since the proofs have only one possible value, runtime behaviour cannot change based on that value. Coq has the type Prop, which can be used to mark proofs as irrelevant at runtime.

But Mathematical Components does not use Prop. It is not clear to me why that is hard.

It is possible to simply replace the data structures with efficient ones, which, given the size of the proof, could be done in reasonable time. Or one could refine (see Section 2.3.1) the functions using CoqEAL to operate on more efficient data structures without changing the proof.

9.2 A verified Round Eliminator

Distributed algorithms researchers can find lower bounds for the time complexity of problems by making the problem easier by allowing strictly more configurations, performing round elimination on the result, then making it easier again, etc. until they arrive at some well-known problem. It can then be argued that the first problem

must be at least as hard as the known problem. Similarly, upper bounds can be proved by making problems harder.

The Round Eliminator software has tools that can be used to quickly perform a series of simplification and round elimination steps. Each one of the tools produces some new problem that has some relationship to the current problem: easier, harder, exactly as hard or exactly one round easier.

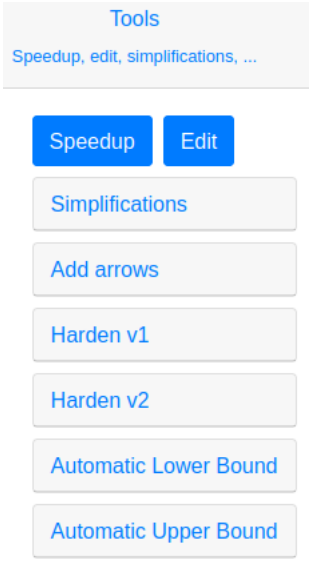


Figure 9: The tools menu of Round Eliminator

Each of the tools could be written as a function in Coq, along with proofs of the relationship of their input and output. With a bit of plumbing, these could be used to write reflective proofs like the following:

```
Lemma a_harder_than_b : time(A) >= time(B).
  add_line [[C, D], [D, E, F]] A.
  speedup.
  eauto using time_gt_trans.
Qed.
```

These could even be automatically generated by Round Eliminator. In cases where Round Eliminator has performed an expensive brute force search, the result of the search can be embedded into the Coq proof in order to avoid running the slower Coq implementations of the tools a lot.

As part of this thesis I have only verified the difficult part of round elimination, maximization. Another possibly difficult part which remains unexplored is formalizing the time complexity of distributed problems. Given a formalization of time complexity, verifying the other tools in Round Eliminator is easy, as they do very simple transformations: adding a line makes the problem easier, removing one makes it harder, renaming a symbol to another makes the problem easier, etc.

Choosing an operation that helps prove a lower bound is nontrivial but that procedure does not need to be proved correct. Were a good series of operations

found, it would just need to be checked with the verified versions of the Round Eliminator tools.

References

[1] *Coq reference manual*, 8.15.1 edition. URL https://coq.inria.fr/refman/.

[2] american fuzzy lop, Aug 2021. URL https://lcamtuf.coredump.cx/afl. [Online; accessed 11. Aug. 2021].

[3] Alkida Balliu, S. Brandt, J. Hirvonen, Dennis Olivetti, Mikaël Rabie, and J. Suomela. Lower bounds for maximal matchings and maximal independent sets. *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 481–497, 2019.

[4] Alkida Balliu, S. Brandt, Yuval Efron, J. Hirvonen, Yannic Maus, Dennis Olivetti, and J. Suomela. Classification of distributed binary labeling problems. In *DISC*, 2020.

[5] S. Brandt. An automatic speedup theorem for distributed problems. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.

[6] S. Brandt and Dennis Olivetti. Truly tight-in-Δ bounds for bipartite maximal matching and variants. *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020.

[7] Mario M. Carneiro. Metamath zero: The Cartesian theorem prover. *ArXiv*, abs/1910.10703, 2019.

[8] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN 0262026651.

[9] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without K. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 257–268, 2014.

[10] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *CPP*, 2013.

[11] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4: 3127–3170, 2015.

[12] Thierry Coquand. A new paradox in type theory. *Studies in Logic and the Foundations of Mathematics*, 134:555–570, 1995.

[13] Lukasz Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61:423 – 453, 2018.

[14] de Ng Dick Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. *Studies in Logic and the Foundations of Mathematics*, 133:73–100, 1970.

[15] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, 2011.

[16] Edward Griffor and Michael Rathjen. The strength of some Martin-Löf type theories. *Archive for Mathematical Logic*, 33(5):347–385, 1994.

[17] Robert Grosse. IntercalScript, Jan 2022. URL https://github.com/Storyyeller/IntercalScript. [Online; accessed 20. Jan. 2022].

[18] J. Hirvonen and J. Suomela. Distributed algorithms 2020. 2020.

[19] Martin Hofmann. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, 1997.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

[21] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, January 2021. doi: 10.5281/zenodo.4457887. URL https://doi.org/10.5281/zenodo.4457887.

[22] Filip Marić. A survey of interactive theorem proving. *Zbornik radova*, 18(26): 173–223, 2015.

[23] Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001.

[24] Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM J. Comput.*, Jul 2006. URL https://epubs.siam.org/doi/10.1137/S0097539793254571.

[25] Dennis Olivetti. Brief announcement: Round eliminator: a tool for automatic speedup simulation. *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020.

[26] Clément Pit-Claudel. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, page 155–174, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381765. doi: 10.1145/3426425.3426940. URL https://pit-claudel.fr/clement/papers/alectryon-SLE20.pdf.

[27] Joonatan Saarhelo. line-combination-proofs, Oct 2021. URL https://github.com/joonazan/line-combination-proofs/tree/master/proofs.

[28] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 59:483–502, 2017.

[29] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages*, 2:1 – 29, 2018.

[30] Mahesh Kumar Thota, Francis H. Shajin, and P. Rajesh. Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering*, 17:331–344, 2020.

[31] László Fejes Tóth. Dichteste kugelpackung. eine idee von gauß. *Abhandlungen der Braunschweigischen Wissenschaftlichen Gesellschaft*, 27:311–321, 1977.

[32] Sebastian Ullrich. Electrolysis reference, Feb 2017. URL http://kha.github.io/electrolysis. [Online; accessed 20. Aug. 2021].

[33] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[34] Théo Zimmermann and Hugo Herbelin. Automatic and transparent transfer of theorems along isomorphismsin the Coq proof assistant. In *CICM*, 2015.