Determination of the fifth Busy Beaver value

The bbchallenge Collaboration* Justin Blanchard Dan Briggs
Konrad Deka Nathan Fenner Yannick Forster Georgi Georgiev (Skelet)
Matthew L. House Rachel Hunter Iijil Maja Kądziołka Pavel Kropitz
Shawn Ligocki mxdys Mateusz Naściszewski savask Tristan Stérin
Chris Xu Jason Yuen Théo Zimmermann

May 12, 2025

Abstract

We prove that $S(5) = 47,176,870$ using the Coq proof assistant. The Busy Beaver value $S(n)$ is the maximum number of steps that an n-state 2-symbol Turing machine can perform from the all-0 tape before halting and S was historically introduced by Tibor Radó in 1962 as one of the simplest examples of a noncomputable function. The proof enumerates 181,385,789 Turing machines with 5 states, and, for each machine, decides whether it halts or not. Our result marks the first determination of a new Busy Beaver value in over 40 years and the first Busy Beaver value to ever be formally verified, attesting to the effectiveness of massively collaborative online research (bbchallenge.org).

Contents

*https://bbchallenge.org

> "In any case, even though skilled mathematicians and experienced programmers attempted to evaluate $\Sigma(3)$ and S(3), there is no evidence that any known approach will yield the answer, even if we avail ourselves of high-speed computers and elaborate programs. As regards $\Sigma(4)$, $S(4)$ the situation seems to be entirely hopeless at present." — Tibor Radó, 1963 [53]

> "*Prediction 5.* It will never be proved that $\Sigma(5) = 4{,}098$ and $S(5) = 47{,}176{,}870$." — Allen H. Brady, 1990 [9]

Figure 1: Tibor Radó, who invented the Busy Beaver game, proved the value of $S(2)$ in 1962 [52], and, was involved in the proof of $S(3)$ in 1963 [41], did not believe that $S(4)$ could be solved at his time. Allen H. Brady, who proved the value of $S(4)$ in 1983 [8], did not believe that $S(5)$ could be solved at all. Amusingly, Brady opens his article on $S(4)$ with Radó's above quote [8]. Under that light, the knowledge of a (small) Busy Beaver value seems to be a reflection of the computing technology available at the time it was proved.

1 Introduction

1.1 Main Result

Are there simple noncomputable functions? What is the *smallest* open problem in mathematics? What do algorithms look like, *in the wild*?

Introduced by Tibor Radó in 1962, *the Busy Beaver game* gives a framework to answer these seemingly independent questions, starting with the first one: Radó's original goal was to "present some very simple instances of noncomputable functions" [52]. The game is as such: (i) run all n-state Turing machines (see Section 2) from the all-zero tape (ii) consider the set of machines that eventually halt (iii) the winner of the game is the halting machine that has the most `1` symbols on its tape when it halts. This maximum number of `1` symbols on final tape among n-state halting machines is called $\Sigma(n)$. Radó also introduced $S(n)$, the maximum number of steps made by a halting n-state Turing machine from all-zero tape.[1] Both functions Σ and S are noncomputable! This is most obvious in the case of S: if an n-state machine runs for more than $S(n) + 1$ steps, we know it will never halt, giving an algorithm to decide Turing's halting problem if S was computable. Because of this tight link between S and the halting problem, we take the liberty to focus our work on S.

While there is no algorithm to compute S for *all* n, we can certainly try to compute S for *some* n. Prior to this work, only the four first values of S had been proved: $S(1) = 1$, $S(2) = 6$ [52], $S(3) = 21$ [41], $S(4) = 107$ [8]. With some early attempts in the 1960s and 1970s, the $S(5)$ quest seriously started in 1983 with a 2-day competition organised at the University of Dortmund[2] with the sole goal of finding new 5-state champions – i.e. 5-state machines achieving better S scores than any previously known machines [44, 46]. The winning machine in Dortmund, found by Schult, achieved $134,467$ steps, establishing $S(5) \geq 134,467$. In 1989, a big step was made when Marxen and Buntrock found a new champion achieving 47,176,870 steps [42], showing $S(5) \geq 47{,}176{,}870$; this machine is given in Figure 2. However, it remained unknown if no other machine could beat it, i.e. whether Marxen and Buntrock's machine was the actual 5-state Busy Beaver winner or not. In 2020, based on the lack of a new 5-state champion in 30 years, Aaronson conjectured that it was the winner, and, $S(5) = 47{,}176{,}870$ [1].

Our main result is to prove this conjecture, using the Coq proof assistant [60], see Theorem 1.1. The Coq proof is called Coq-BB5 and is available at `github.com/ccz181078/Coq-BB5`. We also prove $\Sigma(5) = 4{,}098$, see Section 6, Theorem 6.1.

Theorem 1.1 (Coq-BB5: `Lemma BB5_value`)**.** $S(5) = 47{,}176{,}870$.

The function S can naturally be extended to Turing machines using more than two alphabet symbols [9], like, for instance, $S(2, 3) = 38$ is the value of S for 2-state, 3-symbol machines [9, 45, 32]. We prove, using Coq, that $S(2, 4) = 3{,}932{,}974$, see Theorem 1.2 and Figure 3:

Theorem 1.2 (Coq-BB5: `Lemma BB2x4_value`)**.** $S(2, 4) = 3{,}932{,}974$.

Coq-BB5 provides proofs for $S(5)$ and $S(2, 4)$ — as well as for previously known $S(2)$, $S(3)$, $S(4)$ and $S(2, 3)$. The lists of all the Turing machines evaluated by these proofs are available here.

[1] We avoid using notation BB(n) here as it historically was used to mean Σ [52, 24] and later shifted to mean S [1, 58].
[2] Report of the competition: `https://docs.bbchallenge.org/other/lud20.pdf`.

The goal of this paper is to present these proofs: this paper serves as a "human readable" version of Coq-BB5. As a result of our work, we now have a clearer view of the landscape of small Busy Beaver values, see Table 1.

Symbols	2-State	3-State	4-State	5-State	6-State
2	$S(2) = 6$ [52]	$S(3) = 21$ [41]	$S(4) = 107$ [8]	$S(5) = 47{,}176{,}870$	$S(6) > 10 \uparrow\uparrow 15$
3	$S(2,3) = 38$ [32]	$S(3,3) > 10^{17}$	$S(4,3) > 2 \uparrow\uparrow\uparrow 2^{2^{32}}$	–	–
4	$S(2,4) = 3{,}932{,}974$	$S(3,4) > 2 \uparrow^{15} 5$	–	–	–
5	$S(2,5) > 10 \uparrow\uparrow 4$	–	–	–	–

Table 1: Landscape of small busy beaver values. Cells highlighted in green correspond to values for which we provide Coq proofs. Bright green indicates the new results, $S(5)$ and $S(2,4)$, original to this work. Cells highlighted in orange indicate the existence of a Cryptid (i.e. machines whose halting problem is currently open and believed to be mathematically hard), see Appendix D. Lighter orange means that the existence of a Cryptid comes trivially from reusing a known 3-state 3-symbol Cryptid and ignoring the available additional state or symbol. Lower bounds (see Appendix B), which come from exhibiting Turing machines achieving at least the given step-counts, are expressed using Knuth's "up-arrow notation" which is a way to express iterated exponentiation: $a \uparrow b = a^b$ is exponentiation, $a \uparrow\uparrow b = a^{a^{\cdots^a}}$ is a tower of b powers of a (known as tetration), and higher arrows indicate further levels of iteration.

Challenges. Proving $S(5) = 47{,}176{,}870$ required analyzing the behavior of 181,385,789 Turing machines[3]– evidently requiring computer assistance. The challenge for analysing a halting machine is the case where it halts after a number of steps that is too enormous to be simulated step-by-step (e.g. the current 6-state champion halts after $10 \uparrow\uparrow 15$ steps [31], see Appendix B), this challenge was not met for 5-state machines since they halt in at most 47,176,870 steps – this could not have been known for certain in advance but was believed – which is easy to simulate on modern computers. The challenge for analysing a nonhalting machine is that proving that it does not halt can be hard. How hard?

Dauntingly, any Π_1^0 mathematical statement[4] can be encoded as the halting problem of a Turing machine (from all-zero tape). Such statements are common in mathematics and include famous open problems such as Goldbach's conjecture or the Riemann hypothesis. Goldbach's conjecture, formulated in 1742, is one of the oldest open problems in mathematics and states that "every even positive integer greater than 2 is the sum of two prime numbers". We can build a Turing machine that halts iff the conjecture is false: by enumerating all positive integers, and for each, testing all sums of smaller primes and halting iff we do not find the sum. A machine performing this procedure has been built using only 25 states, and the construction was formally verified using the Lean theorem prover [11, 35, 14].

This means that proving the value of $S(25)$ is at least as hard as solving Goldbach's conjecture for two reasons: (i) impractically, if $S(25)$ was known, witnessing whether the machine halts after $S(25)$ steps would settle the conjecture – this is impractical because $S(25)$ is immensely big, we know that $S(25) > f^2_{\omega^2}(4 \uparrow\uparrow 341)$ where f refers to the *Fast Growing Hierarchy* [2, 69] and (ii) intuitively, the proof of $S(25)$ must contain an argument as of why this particular 25-state machine does not (or, believed less likely, does) halt. Similarly, the Riemann hypothesis has been encoded in a 744-state machine [75, 73, 1]. As low as 15 states is enough to encode a hard number theoretical conjecture by Erdős [57]. Worst, the consistency of common axiomatic systems such as Peano Arithmetic (PA) or Zermelo–Fraenkel set theory (ZF) is also Π_1^0 since one can enumerate proofs in these systems until the proof of a contradiction is found. This has been done in practice for ZF, using 748 states [50, 73, 1, 30]. By Gödel's second incompleteness theorem, this means that proving the value of $S(748)$ cannot be done using ZF. Aaronson conjectures that as low as $S(20)$ cannot be proven in ZF and as low as $S(10)$ cannot be proven in PA [1].

Hence, while 5-state halting machines were not feared, it remained unknown how hard non-halting 5-state machines could get. This article settles the question: the smallest open problem in mathematics (on the Busy Beaver scale) does not arise from 5-state machines – but we have good contenders for 6-state machines, see Section 1.2 and Cryptids (Appendix D).

[3]This exact number can only be known after the proof is done: it is as hard as computing $S(5)$, see Section 3.

[4]A Π_1^0 statement is a statement of first-order logic of the form "$\forall x, \phi(x)$" where ϕ is a sentence using only bounded quantifiers, implying that for a given x, $\phi(x)$ can always be verified by a computer in finite time.

Related Work. In 1983, Allen Brady published the proof of $S(4) = 107$ [8]. One of the proof's main innovations was the introduction of a method to solve the halting problem of a category of machines the author calls *XMas Trees*. A caveat of the proof resides in the following quote from the paper: "All of the remaining holdouts were examined by means of voluminous printouts of their histories along with some program extracted features. It was determined to the author's satisfaction that none of these machines will ever stop." A *holdout* is a machine still needing a proof of halting/nonhalting. Using Coq, we bring additional confirmation that $S(4) = 107$, see Theorem 6.3.

We know of two attempts at solving $S(5)$: in 2003, Georgi Georgiev (Skelet) published the program `bbfind` [20] which enumerates and decides the halting behavior of 5-state Turing machines, claiming to leave unsolved 43 holdouts.[5] Additionally, `bbfind` leaved no 4-state holdouts and computed $S(4) = 107$, agreeing with [8]. However, attesting to the validity of these results is difficult as Skelet's code consisted of ca. 6000 lines of undocumented Pascal code. That said, it turned out to be instrumental to solving $S(5)$ as *bbfind*'s "Closed Position Set" technique (see Section 4.3) was used, simplified, and, improved in order to decide slightly more than 99.87% of the 5-state Turing machines excluding loops (see Section 4.2). Also, all of our 13 *Sporadic Machines*, i.e. machines for which we needed individual proofs of nonhalting, were either among Skelet's 43 holdouts or claimed to have been manually solved by him – Section 5 is dedicated to these longstanding holdouts. Some of Skelet's 43 holdouts were analysed by hand by Briggs starting in 2010 [10]. The second known attempt at solving $S(5)$ was in 2008 with Hertel's "Symbolic induction prover", which claimed to leave only 1,000 holdouts, 900 of which were manually proven not to halt, allegedly leaving only 100 proper holdouts [25]. In an opposite way to Skelet's work, the method is documented but, to the best of our knowledge, no code is made available, nor are the 900 claimed manual proofs, making verification of the result difficult apart from attempting to reproduce from scratch – arguably, verification would still be tedious if the 900 manual proofs were given.

The Busy Beaver problem was also studied in other models of computation than Radó's (c.f. Section 2), including (i) the *quadruple* variation of Turing machines where each transition may move or write a new symbol, but not both [54, 49] (ii) *turmites*, which are Turing machines that operate in 2D [9] and (iii) lambda calculus [67]. For additional historical perspective on the Busy Beaver problem, we refer the reader to Michel's survey and website [46, 44].

Structure of the proof. The proof of our main result, Theorem 1.1, is given in Section 6. The structure of the proof is as follows: machines are enumerated arborescently in *Tree Normal Form* (TNF) [7] – which drastically reduces the search space's size: from about 10 trillion 5-state machines to "only" 181,385,789; see Section 3. Each enumerated machine is fed through a *pipeline* of proof techniques, mostly consisting of *deciders*, which are algorithms trying to decide whether the machine halts or not. Because of the noncomputability of the halting problem, there is no *universal* decider and all the craft resides in creating deciders able to decide large families of machines, in reasonable time. The $S(5)$ pipeline is given in Table 3 – see Table 4 for $S(2, 4)$. Deciders are presented in Section 4. All the deciders in this work were crafted by The bbchallenge Collaboration, see below.

In the case of 5-state machines, 13 *Sporadic Machines* were not solved by deciders and required individual proofs of nonhalting, see Section 5. These machines include suprising behaviours, such as eventually-repeating chaos after more than 10^{51} steps (machine "Skelet #1"), base-Fibonacci double counter (machine "Skelet #10"), or obfuscated Gray code (machine "Skelet #17", [72]) and they are beautiful examples of *algorithms in the wild*: non-human-engineered algorithms that, like deep sea life, were only found by means of exploration. In that spirit, a coarse *zoology* of 5-state Turing machines is proposed in Section 7.

Collaboratively solving the problem: bbchallenge.org. In 2022, Stérin created "The Busy Beaver Challenge", `bbchallenge.org`, an online platform dedicated to collaboratively solving "$S(5) = 47{,}176{,}870$" [58]. Collaboration was motivated by the great amount of Turing machines to study in order to solve the problem. The bbchallenge platform essentially consists of the website, an instant chat *Discord* server[6], and, a wiki.[7] The website serves as an entry-point to the problem and Turing machines visualisation tools both for studying purposes and for piquing the curiosity of visitors with "eye candies". The website was also providing a browsable *seed database*[8] containing a sufficient set of 5-state Turing machines to prove nonhalting in order to solve $S(5)$. Using this database, the task of contributors was to design deciders (see

[5] `https://skelet.ludost.net/bb/nreg.html` and `https://bbchallenge.org/skelet`
[6] `https://discord.gg/wuZhtTvYU3`
[7] `wiki.bbchallenge.org`
[8] `https://github.com/bbchallenge/bbchallenge-seed`

above). For both the seed database and deciders, trust in the results required a strict validation process (formal verification, such as Coq, was only dreamed of when the project started): (i) any algorithm had to be reproduced at least once by an independent contributor, with matching results, (ii) a proof of correctness had to be provided (in natural language, like a regular mathematical article).

With the surprise release of Coq-BB5 in the spring of 2024 (see below), both the seed database and the validation process described above were made obsolete, but almost all the collaborative work performed on `bbchallenge.org` during these 2 years was embedded in the Coq proof – Coq-BB5 also contains many original innovations. Also, although now obsolete, the seed database provided during these 2 years a clear indicator of progress with the number of its machines remaining to be decided, which clearly stimulated collaboration. Deciders that were developed by The bbchallenge Collaboration but that were not used in the Coq proof[9] have been described in [62].

The bbchallenge Collaboration has a hub-and-spoke structure: typically, single contributors or small subsets of contributors made discoveries (mainly, new deciders) and shared their results on the bbchallenge platform (mainly, on our Discord instant chat). Over the span of two years, collaborators organically joined the project and contributed to deciders: more than 20 independent GitHub repositories of deciders[10] were shared on the bbchallenge platform, spanning a vast range of languages – C, C++, Go, rust, Haskell, Coq, Dafny, Lean, Python, PhP, Coq, etc. One core principle of our collaboration was to welcome contributions in any programming language or technology. That way, the use of Coq to solve $S(5)$ was by no mean imposed but came because of the taste and experience of the collaborators who mainly pushed the formal verification effort: mxdys (Coq-BB5) and Kądziołka (`busycoq` [43]).

As mentioned, most of the collaboration was *discussion-driven*, happening continuously, day and night, on our rather entropic instant chat Discord server. At the time of this writing, the Discord server has about 1000 members, of which about 50 are regularly active and about 50,000 messages have been exchanged by about 300 people in total since launch in March 2022. Keeping track of the knowledge produced by the collaboration was a challenge and required dedicated "research maintainers" whose responsibilities very much resembled those of open-source software projects maintainers. We did not have to deal with "trolls" and little moderation was necessary on our channels.[11] Our website bbchallenge.org has had more than 45,000 unique visitors since launched and currently an average of 60 unique visitors per day, with main historical spikes of traffic generated by Hacker News, Quanta Magazine's article and YouTube video about our project, as well as references from blogs[12] and other news report in national journals (France, Austria).

The bbchallenge Collaboration loosely regroups anyone who participated in the discussions across all our channels (Discord chat, forum, wiki, GitHub, emails) which totals hundreds of people; collaborators whose contributions were key to solving $S(5)$ co-author this work and we acknowledge many others, see Appendix A. Some of them are anonymous. We believe that welcoming anonymity participated in having bbchallenge be a place where contributors felt at ease. Most contributors have no academic affiliation and have software engineering related jobs or are students. The community seems to be relatively balanced between three geographical zones: North America, Europe and Asia. A majority of contributors seem to be below the age of 30 but the 30+ age bracket is also well represented. Most contributors never met *In Real Life*. Similarly to the "build in public" philosophy in software, our research happened in public with no retention of information, enabling full reproducibility of the results. Motivated newcomers were able to build on the existent, finding where they wanted to contribute. In contrast, many newcomers also reported being overwhelmed by the entropic nature of our collaboration.

Coq and Coq-BB5. To the best of our knowledge, our work is the first to formally verify a Busy Beaver value. As mentioned above, `bbchallenge` was originally not a formal verification project: formal verification was only dreamed of and most of the work was first conducted using traditional programming languages. Formal verification happened as an unpredicted event, mostly taking everyone by surprise as well as exceeding any expectations we may have had. Importantly, Coq-BB5 is not "just" an *a posterori* formalisation of existing work but also contains many original innovations without which even a nonformalised $S(5)$ proof would have been a lot more complex: for instance, before Coq-BB5 was released, we were lacking simple/elegant proofs for about 2,800 machines.

Our proof, Coq-BB5, is written in Coq[13] [60] which is an interactive theorem prover and programming

[9] Or, in the case of FAR (Section 4.5), only partially used.

[10] Some are listed here: `https://wiki.bbchallenge.org/wiki/Code_repositories`.

[11] The only recurrent moderation effort has been to block bots registering on our wiki to create pages advertising the sale of dubious products.

[12] Mainly `https://scottaaronson.blog/` and `https://sligocki.com`.

[13] Now renamed Rocq.

language based on the Calculus of Inductive Constructions [12] whose development started in 1989. Famous results proven in Coq include the four color theorem on coloring maps [?, ?], Feit-Thompson theorem on odd orders finite groups [?] and Kepler's conjecture on packing spheres [23]. Distinctively, Coq-BB5's objects of study are computer programs (Turing machines) instead of more traditional mathematical objects. Coq-BB5 is available at `https://github.com/ccz181078/Coq-BB5`.

Coq-BB5 implements both the TNF enumeration and the deciders (see proof structure above) directly in Coq, proving them correct, running them and finally using the algorithms' outputs together with the individual proofs for Sporadic Machines to get the result. For instance, proving that a decider is correct amounts to proving that when the decider outputs that a Turing machine halts/does not halt, the machine actually halts/does not halt. Proofs such as Coq-BB5, which rely on computing algorithmic outputs [22, 5], are known as "proofs by reflection" [?] and the Coq proof of the four color theorem also fits in that category [?]. As main output of The Busy Beaver Challenge, the list[14] of all the Turing machines enumerated by the Coq proof, together with their halting status and decider, were *extracted* from the proof using Coq's OCaml extraction capabilities, see Section 6.

With hundreds of millions of Turing machines to study, the use of Coq immensely facilitates scientific consensus on the correctness of the proof: for instance, we are assured that we did not omit any Turing machine in our study. Also, the 13 individual proofs for Sporadic Machines contain technical and error-prone arguments that would be harder to verify and trust if not formalised – e.g. a standalone article was dedicated to machine "Skelet #17" [72] and its Coq proof is about 10,000 lines long, see Section 5.

This totalistic use of Coq in order to solve $S(5)$, where the TNF enumeration of the 181,385,789 Turing machines itself is implemented and ran in Coq, came as a surprise when Coq-BB5 was first released in the spring of 2024 by contributor mxdys who embedded two years of work made by the bbchallenge collaboration as well as improving and introducing new deciders to finish the proof.[15] Indeed, before Coq-BB5, consensus within the bbchallenge collaboration was that formally verifying the TNF enumeration was arduous and, running it, too computationally intensive to be implemented within a proof assistant; at best, the belief was that formal verification efforts would rely on an external, trusted enumeration, such as bbchallenge's seed database (see above). However, formal verification efforts had started as early as 2022 when Fenner verified some deciders using Dafny [47, 34]. Not long after, Kądziołka started verifying deciders in Coq as well as providing together with Yuen 12 out of the 13 individual proofs of nonhalting for Sporadic Machines, see `busycoq` [43], which were reused by Coq-BB5.

Coq-BB5 initially compiled in about 13 hours on a standard laptop but the use of Coq's faster computing engine, `native_compute` [5] and paralelisation of the proof (see Section 3) brought compile time down to about 45 minutes on 13 cores. Trusting that Coq itself is correct, the only elements to check in order to trust Coq-BB5's results are the main theorem statement and the definitions it uses, which have been isolated in an individual file, `BB5_Statement.v`, which is documented with comments, only 121 lines long, and that does not require Coq expertise to be read. Independent Coq experts have reviewed and validated it; they also ruled out the possibility that the proof would try to exploit a potential Coq bug to falsely claim the results.[16] Finally, after compilation, the proof prints the only axiom that it uses, called `functional_extensionality_dep` from Coq's standard library, which claims that two functions are equal if they are equal in all points. This axiom is widely accepted.[17]

1.2 Discussion

Cryptids. The Busy Beaver game is a concrete attempt at identifying the frontier between the knowable and the unknowable: what is the highest n for which we can prove the value of $S(n)$? Knowing that for $n > 5$, as mentioned in Section 1.1, the proof may involve solving arbitrarily difficult problems or worst, simply be outside of PA or ZF.

Concerning difficult problems, in all the next Turing machines classes (orange highlight in Table 1), we have found what we call "Cryptids" which are loosely defined as Turing machines whose halting problem from the all-zero tape is believed to be mathematically hard (Appendix D); for instance, with 6 states, Antihydra (`1RB1RA_0LC1LE_1LD1LC_1LA0LB_1LF1RE_---0RA`) is a machine that does not halt from the all-zero tape if and only if the following conjecture holds:

Conjecture 1.1 (Antihydra does not halt)**.** Consider the Collatz-like map $H : \mathbb{N} \to \mathbb{N}$ defined by $H(x) = 3\frac{x}{2}$ if x is even and $H(x) = 3\frac{x-1}{2}$ if x is odd. Iterating H from $x = 8$, there are never (strictly) more than twice as many odd numbers as even numbers.

[14] Available at: `https://docs.bbchallenge.org/CoqBB5_release_v1.0.0/`.
[15] Coq-BB5's release happened in four stages: (i) 82 holdouts (ii) 1 holdout, Skelet #17 (iii) full proof (iv) optimised proof.
[16] A strong trust indicator is that, in later work, mxdys discovered a consistency bug in the Rocq engine, fixed since then.
[17] It could be removed, at the cost of uunnecessarily complexifying the proof.

Using Conway's terminology, the conjecture *probviously*[18] holds because a probabilistic analysis of the Turing machine suggests that its probability of ever halting is minuscule, namely, $\left(\frac{\sqrt{5}-1}{2}\right)^{1073720885} \approx 4.841 \times 10^{-224394395}$. However, properly proving that Antihydra does not halt is believed mathematically hard, because of resemblance with the notoriously hard Collatz conjecture [33]: Antihydra is a Cryptid. Also, the map H was studied before Antihydra was discovered and is known to not generate Sturmian words [15].

Antihydra is the *smallest*[19] open problem in mathematics, on the Busy Beaver scale. Or, to be exact, one of the smallest since there are other 6-state Cryptids and here is one of them formulated as a Beaver Math Olympiad (BMO) problem:

Problem 1.1 (BMO Problem 1)**.** Let $(a_n)_{n\geq 1}$ and $(b_n)_{n\geq 1}$ be two sequences such that $(a_1, b_1) = (1, 2)$ and

$$(a_{n+1}, b_{n+1}) = \begin{cases} (a_n - b_n, 4b_n + 2) & \text{if } a_n \geq b_n \\ (2a_n + 1, b_n - a_n) & \text{if } a_n < b_n \end{cases}$$

for all positive integers n. Does there exist a positive integer i such that $a_i = b_i$?

This problem is a reformulation of "does `1RB1RE_1LC0RA_0RD1LB_---1RC_1LF1RE_0LB0LE`[20] halt?" – the machine halts if there is i such that $a_i = b_i$. Similarly to Antihydra, the machine is probviously nonhalting[21], but nonetheless, the problem is still open.

We also know of one probviously halting Cryptid: the 3-state 3-symbol machine `1RB2LC1RC_2LC---2RB_2LA0LB0RA` probabilistically has 100% chances of halting[22], and would become the new 3-state 3-symbol champion (considerably extending the current 10^{17} bound) if proven to halt. The apparent contradiction between having 100% chances of halting and not knowing whether it actually halts is solved by thinking of the analogy that there is 100% chances for a random real number chosen between 0 and 1 not to be equal to $\pi/4$ but it can still turn out to be exactly $\pi/4$. Other Cryptids are neither probviously halting or nonhalting, for instance `1RB1LE_0LC0LB_1RD1LC_1RD1RA_1RF0LA_---1RE` is estimated to have 3/5 chances of nonhalting and 2/5 chances of halting.[23]

Cryptids illustrate the facility with which the Busy Beaver game generates small, non-trivial, open mathematical problems, challenging our intelligence and the limits of mathematical knowledge.

Trends in collaborative research. Massively collaborative online research in mathematics and theoretical computer science is a relatively new phenomenon, pioneered by the Polymath Project which collaboratively solved several problems in mathematics [21]. A few differences with `bbchallenge` come to mind such as (i) our use of instant discussion instead of blog comments as main communication media and (ii) the non-academic affiliation of most of our contributors, see Section 1.1, but the essential philosophy of leveraging collective, distributed, intelligence to solve complex problems is the same. In many ways, `bbchallenge`'s effort was much closer to experimental open-source software project developement than to traditional research in mathematics or theoretical computer science: we mainly developed algorithms (deciders) and as a consequence, Coq-BB5 itself is essentially a collection of algorithms, proven correct.

The current increasing popularity of proof assistants such as Coq (Rocq) or Lean within the mathematical and computer science communities naturally accelerates this convergence between collaborative research and opens-source software development: researchers can now leverage the same tools and processes as developers (version control, pull requests, issues, etc.) to massively collaborate on proofs in a scalable way – i.e. not requiring humans to check proofs nor trust.

As a concrete example, shortly after we announced our $S(5)$ result, Tao launched a collaborative pilot project in universal algebra, called the "Equational Theories Project" (ETP), requiring to prove or disprove 22,028,942 implications. The ETP leveraged GitHub and Lean as means of collaboration, a *Zulip* chat server for communication [59, 16, 6]. In contrast with `bbchallenge` which is a rather baroque assembly of many technologies with a "late" use of theorem provers, the ETP focused efforts on formal verification and Lean *by design* and from the start. The project was extremely successful, attracted more than 50 contributors, and was completed in a few months only.[24]

In this context, AI seems to have a bright future: either as an enhanced project knowledge base or as a collaborator itself [64, 71]. Coq-BB5 has started being used as benchmark for AI reasoning [61].

[18] Portmanteau of the words probabilistic and obvious.
[19] We refrained of saying the *simplest* since there are machines with open halting problems which are simpler to solve.
[20] See Section 2 for Turing machines notation.
[21] See probviously nonhalting analysis.
[22] See probvisously halting analysis.
[23] See analysis.
[24] See Tao's personal log.

1.3 Future Work

Progress is ongoing in all next Turing machine classes (orange highlight in Table 1): new deciders are developed to tackle $S(3,3)$, $S(2,5)$ and $S(6)$, including a generalisation of all the regular CTL deciders presented in this paper (see Section 4). Most of these new deciders have been formalised using Coq (Rocq). There currently remains only 6 holdouts for $S(3)$, including the probviously halting suspected new champion (see Section 1.2), and only 83 holdouts for $S(2,5)$ see our lists of holdouts.

Concerning $S(6)$, the TNF enumeration of 6-state machines, which contains about 33 billion machines, has also been partially implemented in Coq (Rocq), and, together with the deciders and about 2,000 individual proofs, "only" about 4,000 holdouts remain. Importantly:

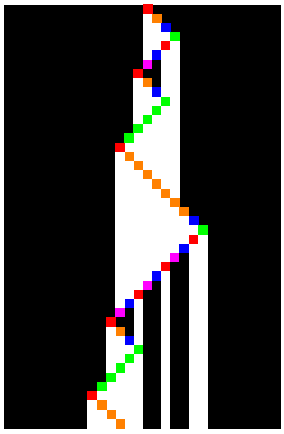
- Antihydra and other Cryptids are among these ca. 4,000 holdouts (see Section 1.2 and Appendix D) which are, most likely, extremely hard problems to solve.
- A halting machine beating the current $10 \uparrow\uparrow 15$ champion (see Table 1 and Appendix B) could be among this ca. 4,000 holdouts which could require significant analysis or accelerated simulators improvements to be detected.

Hence, we can say with absolute certainty that the value of $S(6)$ will never be proved. Nonetheless, `bbchallenge.org` welcomes all new contributors interested in the Busy Beaver game, irrespective of background, favorite programming language or proof assistant (including none)!

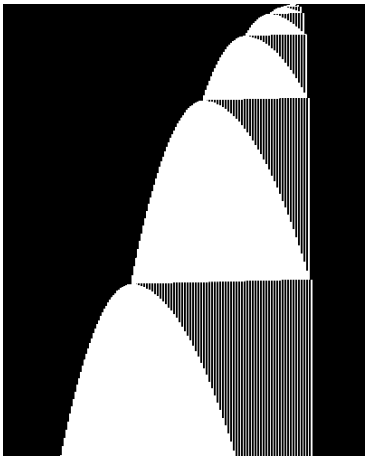
2 Turing machines

	0	1
A	1RB	1LC
B	1RC	1RB
C	1RD	0LE
D	1LA	1LD
E	—	0LA

(a) 5-state 2-symbol Busy Beaver winner. This machine was discovered by Marxen and Buntrock in 1989 [42].



(b) 45-step space-time diagram of the 5-state winner. Head position is coloured to indicate state, see (a).



(c) 20,000-step space-time diagram of the 5-state winner.

Figure 2: Transition table and space-time diagrams of the 5-state 2-symbol busy beaver winner, which halts after 47,176,870 steps. See `1RB1LC_1RC1RB_1RD0LE_1LA1LD_---0LA`.

We consider Turing machines that use a single, discrete, bi-infinite tape – i.e. the tape can be thought as a function $\tau : \mathbb{Z} \to \mathcal{A}$ with $\mathcal{A}$ the alphabet of symbols used by the machine. Machine transitions are either undefined (the machine halts if it ever reaches an undefined transition) or given by (i) a symbol of $\mathcal{A}$ to write (ii) a direction to move (right or left) and (iii) a state to go to – i.e. the transition table of a Turing machine is a partially defined function $\delta : \mathcal{S} \times \mathcal{A} \hookrightarrow \mathcal{A} \times \{\mathrm{L}, \mathrm{R}\} \times \mathcal{S}$, with $\mathcal{S}$ the set of states, e.g. $\{A, B, C, D, E\}$ for 5-state machines. Figure 2(a) gives the transition table of the 5-state 2-symbol Busy Beaver winner. The machine halts after 47,176,870 steps (starting from all-0 tape) when it reads a `0` in state E (undefined transition). Allowing for undefined transitions is a small, consequenceless but useful (see Section 3), deviation from Radó's original setup.

In the Busy Beaver context, machines are always executed from the all-0 tape and starting in state A. Execution goes as follows: at each step, the machine which is in state s looks at which symbol σ is present on the tape cell the head is currently on and then, if defined, executes the instruction given by its transition table, e.g. $\delta(s, \sigma) =$ `0LE` means that the machine will write a `0`, move the cell on the left of the current one and switch to state E. If $\delta(s, \sigma)$ is not defined, the machine halts.

A *configuration* (also known as *execution state*) of a Turing machine is defined by the 3-tuple: (i) state (ii) position of the head on the tape (iii) content of the memory tape. As mentioned above, here, *the initial configuration* of a machine is always (i) state is A, i.e. the first state to appear in the machine's description (ii) head's position is 0 (iii) the initial tape is all-0 – i.e. each memory cell is containing 0. We write $c_1 \to_{\mathcal{M}} c_2$ if a configuration c_2 is obtained from c_1 in one computation step of machine $\mathcal{M}$. We omit $\mathcal{M}$ if it is clear from context. We let $c_1 \to^s c_2$ denote a sequence of s computation steps, and let $c_1 \to^* c_2$ denote zero or more computation steps. We write $c_1 \to \perp$ if the machine halts after executing one computation step from configuration c_1. Halting happens when an undefined machine transition is met i.e. no instruction is given for when the machine is in the state, tape position and tape corresponding to configuration c_1.

Turing machine format. We often communicate Turing machines using the following linear format: `1RB1LC_1RC1RB_1RD0LE_1LA1LD_--0LA` represents the transition table of Figure 2(a) where `_` is used to separate states and transitions are given in read-symbol order. Note that, historically, the undefined transition reached by a halting machine was represented using `1RZ`, hence our format allows to use any letter outside of the state space to represent halting, e.g. `1RB1LC_1RC1RB_1RD0LE_1LA1LD_1RZ0LA`. Multi-symbols machines are represented in the same way, e.g. the 2-state 4-symbol Busy Beaver winner is `1RB2LA1RA1RA_1LB1LA3RB--` (also given by `1RB2LA1RA1RA_1LB1LA3RB1RZ`), see Figure 3. This format can be used as URL on `bbchallenge.org` to display space-time diagrams and known information about the machine, e.g. `https://bbchallenge.org/1RB1LC_1RC1RB_1RD0LE_1LA1LD_---0LA`.

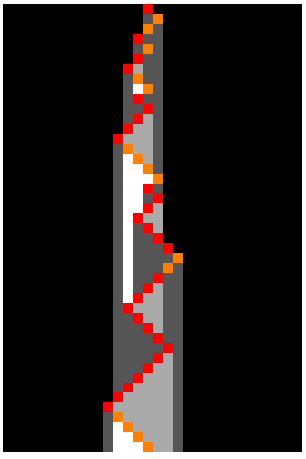
Space-time diagrams. We use space-time diagrams to give a visual representation of the behavior of a given machine. The space-time diagram of machine $\mathcal{M}$ is an image where the i^{th} row of the image gives:

1. The content of the tape after i steps (for 2-symbol machines, black is 0 and white is 1, while for n symbols, black is 0, white is symbol $n-1$ and linear gray-scaling is used in between, e.g. Figure 3).
2. The position of the head is colored to give state information using the following colours for 5-state machines: A, B, C, D, E (you have to look at the row above to deduce what symbol is the head reading, unless it is the initial row, where a `0` is read).

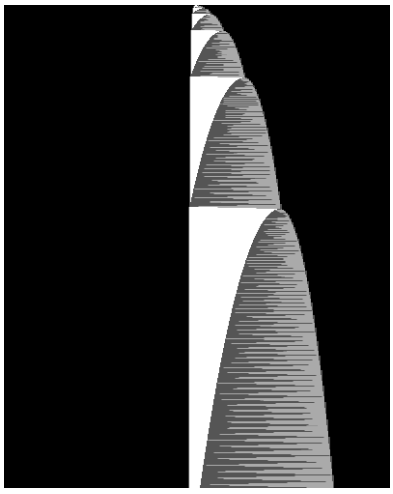
Figure 2(b) gives a 45-step space-time diagram for the 5-state 2-symbol Busy Beaver winner. We often use *zoomed-out* space-time diagrams without state-coloring information, such as Figure 2(c) which gives the 20,000 first steps of the 5-state 2-symbol Busy Beaver winner. Zoomed-out space-time diagrams depicted in this work use a tape of 400 cells unless stated otherwise, the initial cell is generally at the center of the tape but sometimes offset to the right or left. Figure 3 showcases the 2-state 4-symbol Busy Beaver winner.

	0 ■	**1** ■	**2** ■	**3** □
A	1RB	2LA	1RA	1RA
B	1LB	1LA	3RB	—

(a) Transition table of the 2-state 4-symbol Busy Beaver winner found by Ligocki and Ligocki in 2005 [44].



(b) 45-step space-time diagram of the 2-state 4-symbol. Head position is coloured to indicate state, see (a).



(c) 20,000-step space-time diagram of the 2-state 4-symbol winner.

Figure 3: Transition table and space-time diagrams of the 2-state 4-symbol Busy Beaver winner, which halts after 3,932,974 steps. See `1RB2LA1RA1RA_1LB1LA3RB---`.

3 Enumerating Turing machines in Tree Normal Form (TNF)

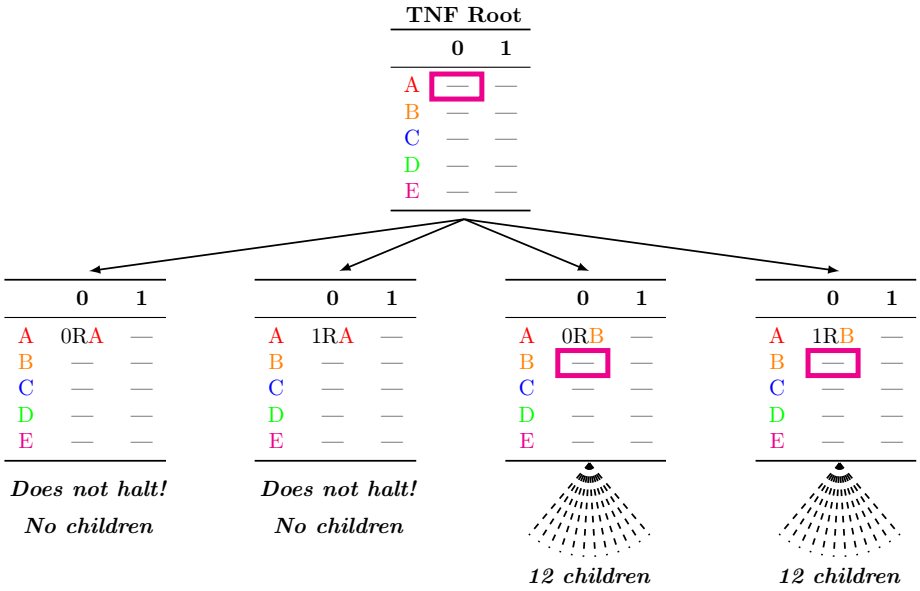


Figure 4: First-level children of the Tree Normal Form (TNF) enumeration of 5-state 2-symbol Turing machines: each node is a Turing machine, nonhalting machines are leaves of the tree. Internal nodes are halting machines, i.e. machines eventually reaching an undefined transition (highlighted in magenta) and their children correspond to all the ways to define this undefined transition. By symmetry, at the first level of the TNF tree, we can ignore machines taking a left move. The two halting machines at the first-level of the tree have 12 children each corresponding to all the choices in $\{0, 1\} \times \{\mathrm{R}, \mathrm{L}\} \times \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ for defining the magenta transition. Note that, in this case, the choice of states is reduced from $\{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$ to $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ in order to prevent constructing machines that are only a permutation of one-another.

Syntactically, as defined in Section 2, there are $(2ns + 1)^{ns}$ Turing machines with n states and s symbols. This gives $21^{10} \simeq 1.67 \times 10^{13} \simeq 16$ trillion possible 5-state 2-symbol Turing machines. However, naively counting Turing machines that way does not account for two phenomenons:

- **Unreachable transitions.** Take the 5-state 2-symbol machine where only the first transition is defined as `0RA` – leftmost machine in Figure 4. This machine is the archetypal Turing machine equivalent of a "while True" infinite loop: the machine will never leave the transition, indefinitely drifting to the right of the tape. Hence, none of the 21^9 machines obtained by defining the other 9 transitions are relevant since these transitions are never reached.

- **State/symbol permutations.** Permuting non-A states and non-zero symbols (A and 0 are special because the initial configuration is all-0 tape in state A) creates identical machines up to renaming, hence studying the halting of only one of them is enough. State/symbol permutation divides the syntactic space size by a factor $(n-1)!(s-1)!$.

Tree Normal Form (TNF) enumeration, introduced by Brady in 1963 in his PhD thesis [7] and illustrated in Figure 4 solves both of these problems: Turing machines are recursively *discovered* starting from the machine with no transitions defined (TNF root). Each enumerated machine is processed by a *pipeline of deciders* (see Section 4) which will either output `HALT`, `NONHALT` or `UNKNOWN` for each machine:

- `HALT`. If the machine halts, such as the rightmost machine in Figure 4, it means that it has met an undefined transition and children of the machine correspond to all the possible ways of defining that undefined transition (highlighted in magenta in Figure 4). Avoiding redundant state/symbol permutations is dealt with at this point by imposing an order on the yet-to-be-seen states/symbols, e.g. children of the rightmost machine in Figure 4 will choose between states $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ instead of $\{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$ since C is the next unseen state.

- `NONHALT`. If the machine does not halt, all its remaining undefined transitions are unreachable and the machine is a leaf of the TNF tree.

- `UNKNOWN`. If the halting status of a machine, it is put in the basket of *holdouts*, i.e. machines that remain to be decided. Having solved $S(5)$ means that there are no more 5-state holdouts.

Hence, by design, TNF enumeration avoids machines with unreachable transitions and state/symbol permutations. One further optimization in the TNF algorithm is, at the first level of the TNF tree (see Figure 4), to avoid machines that make a first move to the left as they can be symmetrised to go to the right instead, e.g. for 5-state 2-symbol machines, this makes the TNF root have 4 children instead of 8. It is also known that only considering machines that first write a 1 is enough to conclude on the value S, but, for simplicity, this is not used in our work [42, 43]. In practice, and in the counts of Table 2, leaves of the TNF tree that have all their transitions defined are not enumerated because they are obviously nonhalting – hence not relevant for computing S.

$S(n)$	Nonhalt	Halt	Total	Syntactic/TNF ratio
$S(2)$	42	19	61	$\times$107 shrink
$S(3)$	3,645	1,772	5,417	$\times$891 shrink
$S(4)$	609,216	249,693	858,909	$\times$8,121 shrink
$S(5)$	133,005,895	48,379,894	181,385,789	$\times$91,958 shrink

Table 2: TNF metrics for $S(2), \ldots, S(5)$: number of nonhalting and halting machines the TNF tree, total number of TNF-enumerated machines and ratio between $(4n+1)^{2n}$ which is the syntactic number of n-state 2-symbol machines and the number of machines in the TNF enumeration.

TNF is unreasonably effective (Table 2): for 5-state 2-symbol machines, it reduces the search-space from 16 trillion to just 181,385,789 machines – a 91,958$\times$ shrink. The variant of TNF that only enumerates machines that start by writing a 1 was implemented (with a step limit instead of deciders) by Heiner and Buntrock in 1989 to find the fifth Busy Beaver winner (Figure 2), taking about 10 days to run at the time [42]. This same variant (using a 47,176,870-step limit, no deciders) was implemented in 2022 for `bbchallenge.org`'s seed database (see Section 1.1), which yielded 88,664,064 holdouts [58]. This database, which had to be trusted, became obsolete with the release of Coq-BB5.

Coq-BB5 TNF implementation. TNF enumeration, as described here, is implemented in Coq-BB5 for the proofs of $S(2), \ldots, S(5)$, see file `TNF.v`. A `SearchQueue` abstraction with DFS capabilities is implemented, see function `SearchQueue_upds`. The search queue is initialised with the TNF root (this is most obvious in the proofs of $S(< 5)$, e.g. see file `BB4_TNF_Enumeration.v`) and deciders (see Section 4) are run on the enumerated Turing machines. Halting machines' children are added to the queue: the goal of the proof is to empty the queue. Importantly, `Lemma SearchQueue_upd_spec` ensures that S can be computed considering only TNF-enumerated machines.

Taking advantage of the tree structure, compilation of the $S(5)$ proof was paralellised by isolating the 12 children of the rightmost machine in Figure 4 in separate, independent files, see folder `BB5_TNF_Enumeration_Roots/`. Parallelising the compilation made the proof compile in 3 hours (on 13 cores) instead of 13 hours. Switching to Coq's more powerful `native_compute` engine [5] further brought parallel compilation time down to 45 minutes. This compilation time could be improved arbitrarily by splitting the tree in even more files with only RAM and number of cores as limitant factors.

Coq-BB5's proof of $S(2,4)$ implements TNF almost exactly in the way described here but for the fact that, for simplicity, it does not impose an order on non-0 symbols meaning that identical machines up to symbol renaming are enumerated. In this "quasi-TNF" setup, the proof of $S(2,4)$ enumerates 2,154,217 machines of which 1,432,880 are nonhalting.

Being able to perform the TNF enumeration of 5-state 2-symbol Turing machines directly in Coq came as a surprise for most `bbchallenge.org` collaborators when Coq-BB5 was released. Results of the enumeration (i.e. list of machines with halting status and decider) were extracted from the proof and made available at `https://docs.bbchallenge.org/CoqBB5_release_v1.0.0/`.

TNF normalisation. To compute the TNF-equivalent of an arbitrary Turing machine, states are reordered by first-visit order, and all R moves are swapped with L (and vice versa) if the initial move is L. This process is not computable in general, as we can't determine in advance which states will be visited. However, for n-state machines, knowing $S(n-2)$ makes TNF normalisation computable.[25]

[25] Apart from always being able to remove unreachable transitions from the initial machine; knowing $S(n)$ solves this.

4 Deciders

4.1 Pipelines and overview

4.1.1 Pipelines

In this work, we call a *decider* a program that takes as input a Turing machine $\mathcal{M}$ and that returns in finite time either `HALT`, `NONHALT`, or, `UNKNOWN` depending on if it was able to detect the machine's halting status or not.[26]

A *pipeline* consists of applying different proof techniques in sequence, mostly consisting of deciders: a machine is tested by each decider successively until one of them outputs `HALT` or `NONHALT`. We call *Sporadic Machines* the 13 machines that were not solved by any decider but using individual proofs of nonhalting instead, see Section 5. Table 3 gives an approximation of the pipeline implemented in Coq-BB5 in order to prove $S(5) = 47{,}176{,}870$, see Theorem 1.1. Similarly, Table 4 and Table 5 respectively give approximations of the pipelines leading to $S(2,4) = 3{,}932{,}974$ and $S(4) = 107$ – the latter confirming the result for $S(4)$ originally given in [8].

The exact pipelines are provided in Appendix C. They differ mainly in the specific parameters and, occasionally, the algorithmic variants used for each decider. In some cases, deciders are interleaved—for example, the loop decider is initially invoked with a small step-count parameter, followed by other deciders, and then called again later with a higher step-count.

$S(5)$ pipeline	Nonhalt	Halt	Total decided
1. Loops, see **Section 4.2**	126,994,099	48,379,711	175,373,810
2. n-gram Closed Position Set (NGramCPS), see **Section 4.3**	6,005,142	0	6,005,142
3. Repeated Word List (RepWL), see **Section 4.4**	6,577	0	6,577
4. Finite Automata Reduction (FAR), see **Section 4.5**	23	0	23
5. Weighted Finite Automata Reduction (WFAR), see **Section 4.6**	17	0	17
6. Long halters (simulation up to 47,176,870 steps)	0	183	183
7. Sporadic machines, individual proofs, see **Section 5**	13	0	13
8. `1RB`-reduction, see **Section 4.1.2**	24	0	24
Total	133,005,895	48,379,894	181,385,789

Table 3: Approximation of the $S(5)$ pipeline as implemented in Coq-BB5. All the 181,385,789 enumerated 5-state machines are decided by this pipeline, which solves $S(5) = 47{,}176{,}870$, see Theorem 1.1. The exact pipeline, with deciders parameters is given in Appendix C.

$S(2,4)$ pipeline	Nonhalt	Halt	Total decided
1. Loops, see **Section 4.2**	1,263,302	721,313	1,984,615
2. n-gram Closed Position Set (NGramCPS), see **Section 4.3**	163,500	0	163,500
3. Repeated Word List (RepWL), see **Section 4.4**	6,078	0	6,078
4. Long halters (simulation up to 3,932,974 steps)	0	24	24
Total	1,432,880	721,337	2,154,217

Table 4: Approximation of the $S(2,4)$ pipeline as implemented in Coq-BB5. All the 2,154,217 enumerated 2-state 4-symbol machines are decided by this pipeline, which solves $S(2,4) = 3{,}932{,}974$, see Theorem 1.2. The exact pipeline, with deciders parameters is given in Appendix C.

$S(4)$ pipeline	Nonhalt	Halt	Total decided
1. Loops, see **Section 4.2**	588,373	249,693	838,066
2. n-gram Closed Position Set (NGramCPS), see **Section 4.3**	20,841	0	0
3. Repeated Word List (RepWL), see **Section 4.4**	2	0	2
Total	609,216	249,693	858,909

Table 5: Approximation of the $S(4)$ pipeline as implemented in Coq-BB5. All the 858,909 enumerated 4-state machines are decided by this pipeline, which brings confirmation that $S(4) = 107$ [8], see Theorem 6.3. The exact pipeline, with deciders parameters is given in Appendix C.

[26] Hence, we don't use the word "decider" in the traditional sense of theoretical computer science since, although our deciders finish, they are partial.

4.1.2 Deciders overview

There are essentially five deciders that were developed to solve $S(5)$, see Table 3: Loops, n-gram Closed Position Set (NGramCPS), Repeated Word List (RepWL), Finite Automata Reduction (FAR) and Weighted Finite Automata Reduction (WFAR). They are individually described in Sections 4.2 to 4.6. All these deciders are orignal.[27] Solving $S(2,4)$ (and previously known $S(4)$ [8]) only used a subset of these deciders (Tables 4) and required a lot less compute and overall effort – e.g. no individual nonhalting proofs, in contrast with 5-state sporadic machines, Section 5.

All these deciders can be expressed in the same general framework, known as Closed Tape Language (CTL) which is attributed to Marxen and Buntrock and was first documented by Ligocki [36]:

General framework: Closed Tape Language (CTL). For a given Turing machine, assume there is a set C of configurations such that:

1. C contains the initial all-0 configuration.
2. C is *closed* under transitions: for any $c \in C$, the configuration one step later belongs to C.
3. C does not contain any halting configuration.

Then, the machine does not halt from any configuration of C and, in particular, from the initial all-0 configuration.

Regularity. All our deciders but WFAR are instances of *regular* CTL, meaning that set C is a regular language – i.e. C is described using a Finite State Automaton (FSA) or, equivalently, a regular expression. Said otherwise, regular CTL approximates the set of configurations of a Turing machine using a regular language that is larger than the machine's set of configurations but on which it is easier (in practice, trivial) to ensure CTL conditions, i.e. (i) membership of the all-0 configuration (ii) closure under Turing machines steps and (iii) absence of halting configurations. NGramCPS and RepWL each focus on specific types of regular languages and are good introductory examples to illustrate this method, while FAR generalises to arbitrary regular languages. We say that a machine is *regular* if it can be proven nonhalting (from all-0 tape) using regular CTL, otherwise we say it is *irregular*. Other regular CTL deciders were developped by The bbchallenge Collaboration, but not used to solve $S(5)$ [27, 17, 18, 62].

Irregularity. WFAR is an analog of FAR, leveraging CTL using *Weighted* FSAs which are a nonregular generalisation of FSAs, see Section 4.6. Not all machines solved by WFAR are necessarily irregular but we strongly suspect that the 17 machines it solves in the $S(5)$ pipeline (Table 3) are irregular because intensive search did not allow finding regular CTL solutions for them. Informal irregularity arguments have been given for sporadic machines "Finned #3" and "Skelet #17" [28, 74], see Section 5.

Interestingly, only regular deciders are needed to solve $S(4)$ and $S(2,4)$, see Tables 4 and 5, which, assuming $S(5)$ irregularity arguments are correct, draws a conceptual line between $S(4)$ and $S(5)$.

Deciders vs. verifiers. While we put all the methods under the decider umbrella it is worthwhile to mention that, in Coq-BB5, only the *verifier* part of FAR and WFAR are implemented. This means that instead of searching for a CTL set C (see above), it is given and verified correct: Coq-BB5 hardcodes a total of 40 FAR / WFAR certificates which are FSAs / Weighted FSAs describing CTL sets C. See files `Verifier_FAR_Hardcoded_Certificates.v` and `Verifier_WFAR_Hardcoded_Certificates.v` in Coq-BB5's folder `BB5_Deciders_Hardcoded_Parameters`.

`1RB`-reduction. Any TNF-enumerated machine (Section 3) whose initial transition writes a `0` and who has at least one transition writing a `1` (TNF guarantees the transition is reachable), can be transformed into a machine that starts by writing a `1` and visits the same configurations (up to state-renaming) but for the first few that wrote a `0`, see Coq-BB5's function `TM_to_NF`. We call this transformation `1RB`-reduction (`1RB` is the first transition of the new machine). Hence, any machine whose reduction to `1RB` already has a proof of nonhalting can be decided using the same argument! This is proven in Coq-BB5's `Lemma TM_to_NF_spec`. For instance, TNF-enumerated machine `0RB0LD_1RC1RE_1LA1RC_1LC1LD_--0RB` reduces that way to sporadic machine "Finned #3" (Section 5), and is decided using the same proof. In the $S(5)$ pipeline (Table 3) this argument is used to decide 24 machines whose `1RB`-reduction correspond to 23 FAR/WFAR certificates (see above) and one sporadic machine, "Finned #3".

[27] There existed a previous algorithm to decide loops [41], but we present a new one.

4.2 Loops

4.2.1 Algorithm

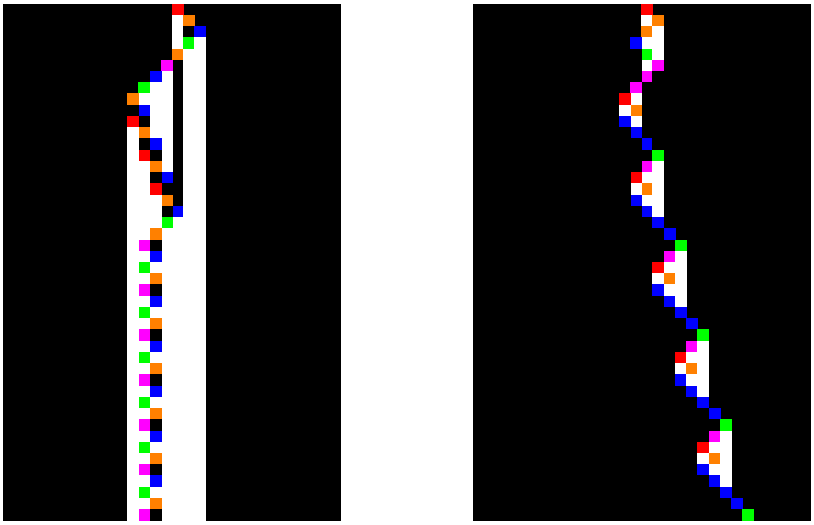


Figure 5: Space-time diagrams of the 30 first steps of a *Cycler*, `1RB---_ORC0LE_1LD0LA_1LB1RB_1LC1RC` (left) and of the 10,000 first steps of a *Translated Cycler*, `1RB---_1LB1LC_ORD0RC_1LE1RE_1LA0LE` (right). Cyclers are machines that eventually repeat the same configuration forever. Translated Cyclers are machines that eventually repeats the same configuration forever, but translated in space. We refer to these two types of machines as *loops*.

The goal of this decider is to recognise two types of Turing machines: (i) *Cyclers* (see Figure 5 left) that eventually repeat the same configuration and therefore loop forever and (ii) *Translated Cyclers* that also eventually repeat the same configuration, but translated in space (see Figure 5 right), analogous to *gliders* in cellular automata. We regroup both types of machines under the umbrella term of *loops*.

Deciding Cyclers reduces to the well-known mathematical problem of detecting the cycles of a function and standard detection algorithms exist [68], the simplest one consisting in memorizing each successive configuration of the machine until encountering one that has been already seen. Translated Cyclers, also known as *Lin's recurrence*, have first been described and decided in Shen Lin's 1963 PhD thesis [41], other similar algorithms to detect them have been developed since then. [28]

Here, we develop a completely different algorithm (Algorithm 1) for deciding both Cyclers and Translated Cyclers at once. The particularity of this algorithm is that it detects loops only by analysing the history of state, read-symbol and head-positions visited by the machine, instead of considering entire configurations (i.e. with full tape content information). Hence, in theory, Algorithm 1 can be implemented to use less memory than previously-known algorithms[29]. This algorithm was introduced by mxdys as part of Coq-BB5.

Let's call the *transcript* of a machine the list of successive state-symbol-pairs visited by the machine from the all-zero tape. For instance, the transcript of the Cycler in Figure 5 (left) starts with `A0 B0 C0 D0 B1 E0 C0 D0 B0 C1 A0` and the transcript of the Translated Cycler in Figure 5 (right) starts with `A0 B0 C0 A0 B1 D1 C1 E0 C0 A1 C1`. Surprisingly, it turns out that in order to detect loops, we only have to track when a transcript repeats the same sequence twice back-to-back, for instance, in the case of the Cycler in Figure 5: `A0 B0 C0 D0 B1 E0 C0 D0 B0 C1 A0 B0 C1 A0 B0 C1 A0 B0 C0 D0 B1 E1 C0 D1 B1 E1 C0 D1`. When such a repetition occurs, we use the extra information of head-position to conclude:

1. If when entering the second repetition the head is at the same position it was at the beginning of the first repetition, then we have detected a Cycler, e.g. for the Cycler in Figure 5 (left), here is the end of the transcript with extra head-position information given after each state-symbol-pair: `B1(-2) E1(-3) C0(-2) D1(-3) B1(-2) E1(-3) C0(-2) D1(-3)`.

[28] https://discuss.bbchallenge.org/t/decider-translated-cyclers/34

[29] In practice, for simplicity, the Coq implementation (see Section 4.2.3) stores entire tapes (whereas it only uses the head's information), hence it does not avail of this potential memory optimisation.

2. If, at the beginning of both repetitions, the head is at the same extremity of the tape (i.e. both positions are either both a local maximum or both a local minimum) then we have detected a Translated Cycler. For the Translated Cycler in Figure 5 (right): `A0(0)* B0(1)* B1(0) C0(-1) D1(0) E1(1)* E1(0) E0(-1) A0(-2) B1(-1) C1(-2) C1(-1) C0(0) D0(1)* E0(0) A0(-1) B1(0) C1(-1) C1(0) C1(1)* C0(2)* D0(3)* E0(2) A0(1) B1(2) C1(1) C1(2) C1(3)* C0(4)*` where `*` indicates steps where the machine is at the right-extremity of the tape (head-position local maximum).

We prove that Algorithm 1 is correct in Theorem 4.5.

Algorithm 1 DECIDER-LOOPS, reformulates the algorithm `loop1_decider` of Coq-BB5.

Input: A Turing machine '$\mathcal{M}$', a step-limit parameter L.
Output: `NONHALT` if the decider detects that the machine is a loop, `HALT` if the machine halts and `UNKNOWN` otherwise.

Simulate $\mathcal{M}$ for L steps and save the history of each consecutive state, read-symbol and position, i.e. consecutive $T_i = (s_i, m_i, d_i) \in \mathcal{S} \times \mathcal{A} \times \mathbb{Z}$ for $0 \leq i \leq L$ and $T_0 = (A, 0, 0)$.

if the machine has halted before L steps **then**
    return HALT

for l **in** $[1, +\infty[$ **do** ▷ l is the length of the potential loop

    if $2l > L$ **then** ▷ The history does not contain two transcript repetitions of size l
        return UNKNOWN

    $K = L - l - 1$
    $o = 0$ ▷ Offset in case we need to keep looking for local-extremum
    allequal = true

    for i **in** $[0, l + o[$ **do**
        if $K - i < 0$ **then**
            break

        $s, m, d = T_{L-i}$
        $s', m', d' = T_{K-i}$

        if $s \neq s'$ **or** $m \neq m'$ **then** ▷ Comparing state-symbol-pair equality at each step
            break

        if $i = l + o - 1$ **then**
            if $d = d'$ **then**
                return `NONHALT` ▷ We've detected a Cycler

            if $d = \max\{d_j \mid j < L - i\}$ **and** $d' = \max\{d_j \mid j < K - i\}$ **then**
                return `NONHALT` ▷ We've detected a (positive) Translated Cycler

            if $d = \min\{d_j \mid j < L - i\}$ **and** $d' = \min\{d_j \mid j < K - i\}$ **then**
                return `NONHALT` ▷ We've detected a (negative) Translated Cycler

            $o = o + 1$

return `UNKNOWN`

4.2.2 Correctness

Proving the correctness of this decider is surprisingly nontrivial. In this work, $\mathbb{N} = \{0, 1, \dots\}$ and $\mathbb{N}^+ = \{1, 2, 3 \dots\}$. Let's represent the space-time diagram of a given Turing machine $\mathcal{M}$ (Section 2) using partially defined functions $f_\mathcal{M}, g_\mathcal{M}, h_\mathcal{M}$, and, $F_\mathcal{M}$:

$$f_\mathcal{M} : \mathbb{N} \hookrightarrow \mathbb{Z} \to \mathcal{A}, \text{ tape content}$$
$$g_\mathcal{M} : \mathbb{N} \hookrightarrow \mathbb{Z}, \text{ head position}$$
$$h_\mathcal{M} : \mathbb{N} \hookrightarrow \mathcal{S}, \text{ head state}$$
$$F_\mathcal{M} : \mathbb{N} \hookrightarrow \mathbb{Z} \to (\mathcal{A} \times \mathcal{S}), \text{ tape content and head state}$$

At time $t \in \mathbb{N}$, $f_\mathcal{M}(t)$ gives the tape content (as a total function $\mathbb{Z} \to \mathcal{A}$), $g_\mathcal{M}(t)$ gives the head position and $h_\mathcal{M}(t)$, the head state, assuming in each case that $\mathcal{M}$ has not halted before time t, otherwise values are not defined. For brevity, when the Turing machine is clear from context, we may write f, g, h. In the case of f, we will also use notation $f(t, z)$ or $f(p)$ with $p \in \mathbb{N} \times \mathbb{Z}$ seen as a vector, instead of $f(t)(z)$ when $f(t)$ is defined. Using same extended notations as for f, we also define $F_\mathcal{M}(t, z) = (f(t, z), h(t))$ which gives tape content with head state information at each time when f and h are defined. Finally, when assuming/claiming $f(x) = f(y)$ for some $x, y \in \mathbb{N} \times \mathbb{Z}$ we also implicitly assume/claim that f is defined at x and y; same for F, g, and, h.

Transcripts as defined in Section 4.2.1 correspond to sequences of the form $F_\mathcal{M}(t, g(t))$ with $t \in \mathbb{N}$.

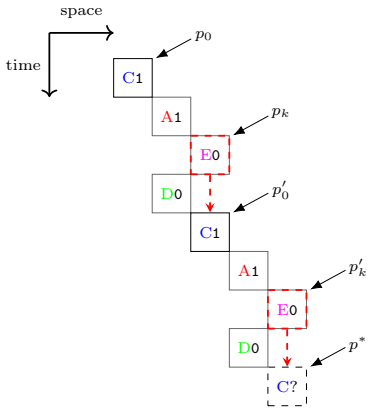


Figure 6: Illustration of Lemma 4.1: head-only space-time diagram showing transcript repetition C1 A1 E0 D0 C1 A1 E0 D0. Coordinates $p_0 = (t_0, g(t_0))$ correspond to the beginning of the first repetition, and $p'_0 = (t'_0, g(t'_0))$ of the second. At position p^*, we can easily show that the machine shares same state, C, than at position p'_0, Lemma 4.1 Point 2. Less immediate, in this case, using p_k and p'_k, we can show that positions p^* and p'_0 also share same read-symbol (depicted as ? to signify that it is less immediate to show), which is the symbol outputted after p_k and p'_k, Lemma 4.1 Point 4.

Lemma 4.1. Let $\mathcal{M}$ be a Turing machine. Assume there is $t_0 \in \mathbb{N}$ and $l \in \mathbb{N}^+$ such that for all $0 \leq i < l$:

$$F_\mathcal{M}(p_i) = F_\mathcal{M}(p'_i)$$

with $t_i = t_0 + i$, $t'_i = t_i + l$ and $p_i = (t_i, g(t_i))$, $p'_i = (t'_i, g(t'_i))$. Call $p^* = (t^*, g(t^*))$ with $t^* = t'_0 + l$. Then:

1. For all $0 \leq i < l$ we have: $p'_i - p_i = p'_0 - p_0$. We also have $p^* - p'_0 = p'_0 - p_0$.
2. We have $h(t^*) = h(t'_0)$.
3. For all $0 \leq i < l$, $g(t_i) = g(t'_0) \Leftrightarrow g(t'_i) = g(t)$.
4. If there is $0 \leq k < l$ such that $g(t'_k) = g(t^*)$ then $f(p^*) = f(p'_0)$.

Figure 6 illustrates this Lemma and Point 4 in particular.

Proof. First note that by definition, all p_i and p'_i correspond to head positions and the condition $F_\mathcal{M}(p_i) = F_\mathcal{M}(p'_i)$ means that there is a repetition in the machine's transcript (see Section 4.2.1).

1. Because $F(p_0) = F(p'_0)$ we know that at times t_0 and t'_0 the head is in same state, reading the same symbol, hence the same transition executes and, in particular, both heads move in the same direction $m \in \{-1, 1\}$, giving the existence of $u = (1, m)$ such that $p_1 = p_0 + u$ and $p'_1 = p'_0 + u$. Hence, $p'_1 - p_1 = p'_0 + u - p_0 + u = p'_0 - p_0$. Repeating the same argument for each $i < l$ gives $p'_i - p_i = p'_0 - p_0$. Finally, applying the same argument with $F(p_{l-1}) = F(p'_{l-1})$ gives $p^* - p'_0 = p'_0 - p_0$ as p^* corresponds to one time step after p'_{l-1} and p'_0 one time step after p_{l-1}.

2. Similarly to above, $F(p_{l-1}) = F(p'_{l-1})$ implies that the machine will transition to the same state after p_{l-1} and p'_{l-1}, giving $h(t^*) = h(t')$.

3. Let $0 \leq i < l$ such that $g(t_i) = g(t'_0)$. Using Point 1, we have $p'_i = p_i + (p'_0 - p_0)$, meaning $g(t_i) = g(t'_0) \Leftrightarrow g(t'_i) = g(t_i) + (p'_0 - p_0)$ which rewrites as $g(t_i) = g(t'_0) \Leftrightarrow g(t'_i) = g(t'_0) + (p'_0 - p_0)$. Point 1 again with $p^* = p'_0 + (p'_0 - p_0)$, we get $g(t'_0) + (p'_0 - p_0) = g(t)$ and $g(t_i) = g(t'_0) \Leftrightarrow g(t'_i) = g(t)$ as needed.

4. Without loss of generality, let's assume that k is maximal. Using Point 3 we get that $g(t_k) = g(t'_0)$ and, by maximality of k, there is no $s > k$ with $s < l$ such that $g(t_s) = g(t'_0)$. Because $F(p_k) = F(p'_k)$, the same symbol is outputted after p_k and p'_k, giving $f(t_k + 1, g(t_k)) = f(t'_k + 1, g(t'_k))$ and by maximality ok k, the cells are not revisited respectively before p_0 and p^*, giving $f(p'_0) = f(t_k + 1, g(t_k))$ and $f(p) = f(t'_k + 1, g(t'_k))$. Hence we have $f(p^*) = f(p'_0)$ as needed.

□

Definition 4.2 (Loops)**.** Let $\mathcal{M}$ be a Turing machine. Let $l \in \mathbb{N}^+$, $t_0 \in \mathbb{N}$ and $p_i = (t_0 + i, g(t_0 + i))$ for $0 \leq i < l$. We say that $\mathcal{M}$ is a *loop* of period l and pre-period t_0 if for all $t > t_0$, $F_{\mathcal{M}}(t, g(t)) = F_{\mathcal{M}}(p_j)$ with $j = t - t_0 \bmod t'_0 - t_0$.

Lemma 4.3. Loops do not halt.

Proof. This is immediate, by Definition 4.2, the space-time diagram of a loop is infinite, the machine does not halt. □

Theorem 4.4 (Loops)**.** Let $\mathcal{M}$ be a Turing machine. Assume there is $t_0 \in \mathbb{N}$ and $l \in \mathbb{N}^+$ such that for all $0 \leq i < l$:

$$F_{\mathcal{M}}(p_i) = F_{\mathcal{M}}(p'_i)$$

with $t'_0 = t_0 + l$ and $p_i = (t_0 + i, g(t_0 + i))$ and $p'_i = (t'_0 + i, g(t'_0 + i))$. Then, three cases:

1. If $g(t'_0) = g(t_0)$, then $\mathcal{M}$ is a loop, more specifically called a *Cycler*.

2. If $g(t'_0) \geq g(t_0)$ and the tape content to the right of p_0 is the same as to right of p'_0, i.e. $\forall z \geq 0 \in \mathbb{Z}\ f(t_0, g(t_0) + z) = f(t'_0, g(t'_0) + z)$, then $\mathcal{M}$ is a loop, more specifically called a (positive) *Translated Cycler*.

3. If $g(t'_0) \leq g(t_0)$ and the tape content to the left of p_0 is the same as to left of p'_0, i.e. $\forall z \leq 0 \in \mathbb{Z}\ f(t_0, g(t_0) + z) = f(t'_0, g(t'_0) + z)$, then $\mathcal{M}$ is a loop, more specifically called a (negative) *Translated Cycler*.

In these three cases, $\mathcal{M}$ has period l and the pre-period t_0, and, does not halt.

Proof. Consider $p^* = (t^*, g(t^*))$ with $t^* = t'_0 + l$ which corresponds to one time step after p'_{l-1}. Figure 7 illustrates the situation for the theorem's cases 1 (on the left) and 2 (on the right): p^* is the coordinates of the black dashed cell.

We show that $F(p) = F(p'_0)$. By Lemma 4.1 Point 2, we know that $h(t^*) = h(t'_0)$ hence we have to show that the read symbol at p^* and p'_0 are also the same, i.e. $f(p) = f(p'_0)$. Three cases:

1. Case $g(t'_0) = g(t_0)$, illustrated in Figure 7 (left). By Lemma 4.1, Point 1, we know that $p^* - p'_0 = p'_0 - p_0$. Because $g(t'_0) = g(t_0)$, the space component of $p^* - p'_0$ is 0 and we have that $g(t^*) = g(t'_0)$. Hence, we know that the cell at tape position $g(t^*)$ has been visited at least once between time steps t'_0 and $t'_0 + l - 1$, which, by Lemma 4.1, Point 4, gives that $f(p^*) = f(p'_0)$.

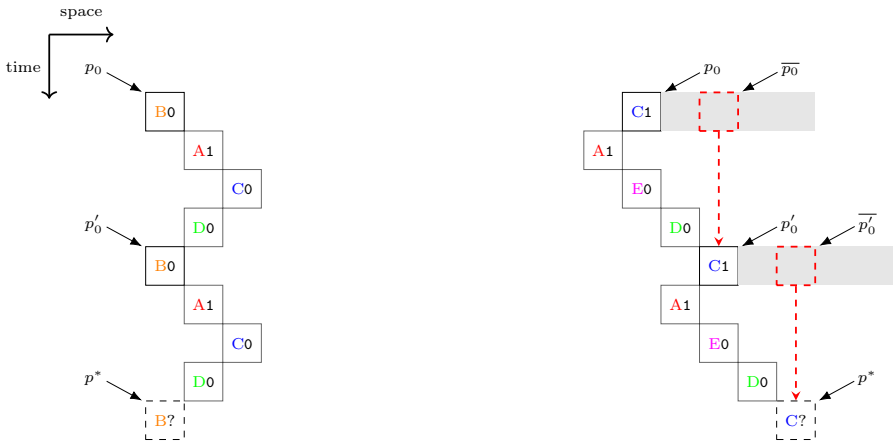


Figure 7: Illustration of Theorem 4.4: Case 1, Cycler (left) and Case 2, positive Translated Cycler (right). In both cases, we show a head-only space-time diagram with one transcript repetition (see Section 4.2.1). Coordinates $p_0 = (t_0, g(t_0))$ correspond to the beginning of the first repetition, and $p'_0 = (t'_0, g(t'_0))$ of the second. In both case, showing that cell at position p^* shares the same state as cell at position p'_0 is rather easy (Lemma 4.1, Point 2) while showing that they share same read-symbol (depicted as `?`) requires more work, Theorem 4.4. In the case of Translated Cyclers, this is either done using Lemma 4.1, Point 4, depicted in Figure 6 or, using the assumption that the tape content after p_0 and p'_0 is the same, here symbolised using grey shading (right). In that case, using $\overline{p_0}$ and $\overline{p'_0}$ we show that the read-symbol at p^* is the same that at $\overline{p'_0}$.

2. Case $g(t'_0) > g(t_0)$, illustrated in Figure 7 (right). If there is a time step between t'_0 and $t'_0 + l - 1$ such that tape position $g(t^*)$ has been visited, by Lemma 4.1, Point 4, we get that $f(p^*) = f(p'_0)$.

   If there is no such time step, we have $f(p^*) = f(\overline{p'_0})$ with $\overline{p'_0} = (t'_0, g(t))$, dashed red in Figure 7 (right). By Lemma 4.1, Point 3, we also have that there is no time step between t_0 and $t_0 + l - 1$ such that tape position $g(t'_0)$ has been visited. Hence, we get $f(p'_0) = f(\overline{p_0})$ with $\overline{p_0} = (t_0, g(t'_0))$. By hypothesis, tape content to the right of p_0 is the same as to the right of p'_0, and $p^* - p'_0 = p'_0 - p_0$ (Lemma 4.1, Point 1) implies that $g(t^*) - g(t'_0) = g(t'_0) - g(t_0)$, hence $f(\overline{p_0}) = f(\overline{p'_0})$ and, finally, $f(p^*) = f(p'_0)$.

3. Case $g(t'_0) < g(t_0)$, handled symmetrically to 2.

From there, we get $F(p^*) = F(p'_0)$, and the argument can be repeated starting with $p_1, \ldots, p_{l-1}, p'_0$ acting as new $p_0, \ldots, p_{l-1}$ and $p'_1, \ldots, p'_{l-1}, p^*$ as new $p'_0, \ldots, p'_{l-1}$ by noting that for cases 2 and 3, the fact that the tape is the same after p_0 and p'_0 implies that it is also the same after p_1 and p'_1. Hence, inductively, $\mathcal{M}$ is a loop and, by Lemma 4.3, it does not halt. □

Theorem 4.5 (Coq-BB5: `Lemma loop1_decider_WF`)**.** Let $\mathcal{M}$ be a Turing machine and $L \in \mathbb{N}^+$ a step-limit. DECIDER-LOOPS($\mathcal{M}$, L) terminates and its result is correct – see Algorithm 1:

- If the result is `HALT` then $\mathcal{M}$ halts from the all-zero tape.
- If the result is `NONHALT` then $\mathcal{M}$ does not halt from the all-zero tape.

Proof. The call to DECIDER-LOOPS($\mathcal{M}$, L) terminates because of Algorithm 1, l.12. The call returns `HALT` if and only if $\mathcal{M}$ halts within L steps from the all-zero tape, see Algorithm 1, l.7, hence if the call returns `HALT` we know that the machine halts. The interesting case is the loop-detection leading to `NONHALT`.

Algorithm 1 finds t_0 and l satisfying the hypotheses of Theorem 4.4: $t_0 = K - l - 1 - o$ and where $F_{\mathcal{M}}(p_i) = F_{\mathcal{M}}(p'_i)$ is guaranteed thanks to Algorithm l.25. Theorem 4.4 Cases 1, 2, and, 3 are respectively handled by Algorithm l.29, l.32, and, l.35. In Case 2/Case 3, the condition of having tapes be the same to the right/left of p_0 and p'_0 is handled by making sure the head is at the maximum/minimum seen position of the tape in both cases, ensuring that there are only 0s to the right/left, and therefore satisfying the condition. Hence, we get that $\mathcal{M}$ does not halt. □

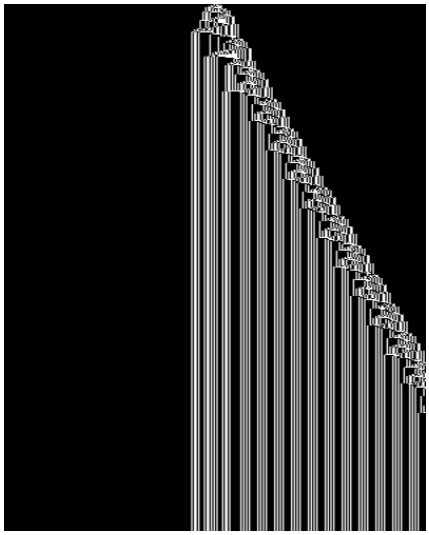


Figure 8: 10,000-step space-time diagram of a Translated Cycler not decided by the decider for loops in Coq-BB5 (it is decided by NGramCPS, see Section 4.3). See `1RB0LE_1LC0RD_---1LD_1RE0LA_1LA0RE`.

4.2.3 Implementations and results

Step-limit parameter L	Nonhalt	Halt	Total decided
130	126,950,828	48,367,435	175,318,263
4100	43,269	12,276	55,545
1,050,000	2	0	2
Total	126,994,099	48,379,711	175,373,810

Table 6: Machines decided by using the loop deciders (Algorithm 1) in the $S(5)$ pipeline (Table 3) per step-limit parameter L.

The decider for loops, Algorithm 1, is implemented as part of Coq-BB5 (function `loop1_decider`). As advertised in the $S(5)$ pipeline (Table 3), it decides a very important proportion the enumerated 5-state Turing machines: 95.48% of the nonhalting machines and more than 99.99% of the halting ones and this with fairly low step-limit parameters, see Table 6. This means, for instance, that 99.99% of the enumerated 5-state halting machines halt before $4,100$ steps.

The number of nonhalting machines decided this decider in Coq-BB5 (i.e. 126,994,099, see Table 6) is a lower bound of the actual number of 5-state loops. Two examples:

1. Figure 8 gives a Translated Cycler that is decided by the n-gram Closed Position Set (NGramCPS) decider, see Section 4.3, this is because higher step-limit L would have been needed to be detected by Algorithm 1.

2. Infamously, the Sporadic machine (i.e. machine which required an individual proof of nonhalting) named "Skelet #1", see Section 5, is a Translated Cycler but with enormous parameters: it does not start looping before 5.41×10^{51} steps and has a period of more than 8 billion steps [38]. There is no reasonable step-limit L for which this machine would have been decided by the decider for loops, neither in fact by any of the deciders presented in this work which all all more or less rely on step-by-step simulation. An individual proof of nonhalting was required [43].

In this sample of 126,994,099 nonhalting loops we find approximatively 86% Translated Cyclers and 14% Cyclers which suggests that, in general, Translated Cyclers are much more common than Cyclers.

Other implementation. Algorithm 1 also has a Python implementation.[30]

[30] https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-loops-reproduction

4.3 n-gram Closed Position Set (NGramCPS)

The n-gram Closed Position Set (NGramCPS) decider which we introduce here is a simplification of an earlier technique, Closed Position Set (CPS), itself introduced in `bbfind` [20], see Section 1.1. Surprisingly, NGramCPS is a relatively simple technique which makes a potent decider as it decides 99.89% of all nonhalting enumerated 5-state machines excluding loops, see Table 3.

The method is especially potent when augmenting the binary alphabet of Turing machines to record extra information on the tape, such as a fixed-length history of previously seen (state,symbol) pairs, see Section 4.3.2.

Coq-BB5's NGramCPS core algorithm, without augmentations, was first developed in the Rust programming language by bbchallenge's contributor Nathan Fenner [48]. Augmentations were introduced by mxdys as part of Coq-BB5.

4.3.1 Algorithm

Algorithm 2 gives a pseudo-code of the NGramCPS decider. The decider considers finite, *local configurations* of a Turing machine consisting of: (i) the *n-grams* (see after) respectively to the left and to the right of the head, (ii) the state the machine is in, and (iii) the symbol currently read by the head, referred to as *middle* symbol (as opposed to the left and right part of the tape, modelled by the n-grams). By *n-gram*, we mean a sequence of $n > 0$ symbols from the tape alphabet (for instance, the binary alphabet $\mathcal{A} = \{0, 1\}$).

The algorithm builds a set of local configurations *potentially* reachable by the machine until either an undefined transition is met (Algorithm 2, l.16) or no new configurations are added to the set, i.e. the set is closed under Turing machine operations (Algorithm 2, l.41). In the first case, the decider cannot conclude and the machine is left undecided. In the second case, the decider concludes that the machine does not halt as no undefined transition (i.e. where the machine could be asked to halt) can be reached.

The central idea of this decider and the reason behind using the "n-gram" terminology (originating from *n-gram models* in language analysis) is better illustrated by the following example. Let $n = 3$ and consider local configuration `011 [B0] 100`, meaning that the left n-gram is `011`, right n-gram is `100`, the machine is in B and reading symbol `0`. Assume that the machine's transition for reading a `0` in state B is `1RC`, meaning that the machine writes `1`, moves right and transitions to state C. The local configuration becomes `011 1 [C1] 00?`, where `?` means that we don't know which symbol to use. Then:

1. **Left n-gram update.** We record the left n-gram `011` as seen (it is inserted in set L, Algorithm 2, l.19) and we discard its first bit, updating the left n-gram to `111`. The local configuration becomes `111 [C1] 00?`.

2. **Right n-gram update.** In order to deal with the unknown symbol `?`, we look among the previously seen right n-grams (contained in set R in Algorithm 2) the ones that start by `00`. For instance, let's assume it is `000` and `001`. Then we add both local contexts `111 [C1] 000` and `111 [C1] 001`, if not already in: (a) to our set of local configurations (Algorithm 2, l.25), and (b) to our set of configurations to visit (Algorithm 2, l.26) in order to repeat this procedure (or symmetrical when the machine moves left) on them.

The algorithm systematically revisits all previously added local configurations, in case they contain a right/left n-gram that was newly met (Algorithm 2, l.7). Assuming a finite tape alphabet (which we always do in this work), the algorithm will eventually terminate since the number of possible local configurations is finite. In practice, one may add a limit on the number of iterations to avoid long computations.

Theorem 4.6 (Coq-BB5: `Lemma NGramCPS_decider_spec`)**.** Let $\mathcal{M}$ be a Turing machine using tape alphabet $\mathcal{A}$ containing zero symbol $\mathcal{A}_0$ and let $n \in \mathbb{N}^+$ be the n-gram length parameter. DECIDER-NGRAMCPS($\mathcal{M}$, $\mathcal{A}_0$, n) terminates and its result is correct – see Algorithm 2: if it returns `NONHALT` then $\mathcal{M}$ does not halt from the all-$\mathcal{A}_0$ tape.

Proof. Algorithm 2 is guaranteed to terminate because either an undefined transition is eventually met (Algorithm 2, l.16) or because the set of local configuration – which is bounded by the finite set of all possible local configurations – is saturated (Algorithm 2, l.41).

By construction, Algorithm 2 overestimates the set of all local configurations reached by the machine from the all-$\mathcal{A}_0$ tape, i.e. it contains at least all the reached local configurations and potentially more. If this set contains no local configuration leading to an undefined transition, we are assured that the machine does not halt, Algorithm 2, l.41. □

4.3.2 Tape alphabet augmentations

The NGramCPS decider becomes particularly powerful for deciding 5-state 2-symbol Turing machines when augmenting the 2-symbol alphabet to store more information on the tape. Two augmentations are used in Coq-BB5:

1. **Fixed-length history.** In this variant, tape symbols encode the current binary symbol on a cell as well as a fixed-length list of previously seen (state,binary symbol) pairs seen on the cell. For instance, if the non-augmented machine currently reads binary symbol `1` and the machine has previously visited the cell in state A reading symbol `0` and before that in state B reading symbol `1`, in the augmented machine, the cell will contain the augmented symbol "`1, [(A,0),(B,1)]`". If the history length is set to 2 and the machine was in state C when reading "`1, [(A,0),(B,1)]`" the cell will be updated to "`0, [(C,1),(A,0)]`", assuming the transition of the machine for reading a `1` in state C requires to write symbol `0`. The zero-symbol for this augmentation $\mathcal{A}_0$ is "`0, []`". Furthermore, it is easy to verify that if the decider returns `NONHALT` for a fixed-length augmented machine, then the non-augmented machine does not halt.

2. **Least Recent Usage history (LRU).** In this variant, tape symbols encode the set of state-symbol pairs seen at that cell, in order of when it was seen last, the most recent first. For instance, if the non-augmented machine currently reads binary symbol `1` and the machine has previously visited the cell in state D reading symbol `1` and before that in state D reading symbol `0` and before that in state C reading symbol `1`, in the augmented machine, the cell will contain the augmented symbol "`1, [(D,1),(D,0),(C,1)]`". If the machine was in state C when reading "`1, [(D,1),(D,0),(C,1)]`" the cell will be updated to "`0, [(C,1),(D,0),(D,1)]`", (assuming the transition of the machine for reading a `1` in state C writes symbol `0`) with pair `(C,1)` bubbling up to the beginning of the LRU history. The zero-symbol for this augmentation $\mathcal{A}_0$ is also "`0, []`". Similarly to above, one can verify that if the decider returns `NONHALT` for an LRU augmented machine, then the non-augmented machine does not halt. One fundamental difference with the fixed-length history augmentation is that here, the history is not of fixed length but is bounded by number of states times number of symbols, i.e. 10 in the case of $S(5)$.

4.3.3 Implementations and results

Variant	Nonhalt
NGram-CPS without augmentation	5,117,863
NGram-CPS augmented using fixed-length history	887,093
NGram-CPS augmented using Least Recent Usage history	182
Total decided	6,005,138

Table 7: NGramCPS results in the $S(5)$ pipeline (see Table 3) per variant (see Section 4.3.2).

Coq-BB5 implements NGramCPS (Algorithm 2) in the three variants discussed here, (i) without augmentation (function `NGramCPS_decider_impl2`) – i.e. using standard binary alphabet $\mathcal{A} = \{0, 1\}$ (ii) fixed-length history (function `NGramCPS_decider_impl1`) and (iii) Least Recent Usage history (function `NGramCPS_LRU_decider`). Compared to Algorithm 2, Coq-BB5 implementations integrate an additional *gas parameter* allowing them to terminate early for the sake of performance. The implementations for (ii) and (iii) use the same core implementation as for (i) just accordingly augmenting the tape-alphabet of the machine and its read/write behavior (see definitions `TM_history` and `TM_history_LRU`).

Altogether, NGramCPS decides 99.89% of all nonhalting enumerated 5-state machines excluding loops, see Table 3. The number of machines decided by each NGramCPS variant in the $S(5)$ pipeline (Table 3) are given in Table 7. Augmentations allowed to decide machines that resisted all other methods, without having to resort to individual proofs of nonhalting – see details in the full $S(5)$ pipeline, Appendix C.

Figure 9 gives an example of a "fractal-looking" 5-state Turing machines that is solved by the LRU augmentation but has no known solution with standard NGramCPS or the fixed-length history augmentation.

Algorithm 2 DECIDER-NGRAMCPS

Input: A Turing machine $\mathcal{M}$, the zero symbol of the alphabet $\mathcal{A}_0$, the size of the n-grams $n > 0$.
Output: `NONHALT` if the decider detects that the machine doesn't halt and `UNKNOWN` otherwise.
$g_0 = (\mathcal{A}_0)^n$ ▷ The zero n-gram consists of n zero symbols
$L = \{g_0\}$ ▷ The seen left n-grams
$R = \{g_0\}$ ▷ The seen right n-grams
$C = \{\{$.left $= g_0$, .right $= g_0$, .state $=$ A, .middle $= \mathcal{A}_0$ $\}\}$ ▷ The seen local configurations
while true **do**
    $V = C$
    any_updates = false
    while $|V| \neq 0$ **do**
        $c = V$.**pop**()
        $c' = c$
        $\{w, d, s\} = \mathcal{M}$(c.state, c.middle) ▷ Transition's write symbol, move direction, and next state

        if s is undefined **then** ▷ Undefined transition is met, we cannot conclude
            return UNKNOWN

        if d is Right **then**
            Insert c.left in L
            Set c'.left to the last $r-1$ symbols of c.left followed by w
            Set c'.middle to the first symbol of c.right
            for each ngram $r \in R$ starting with the last $r-1$ symbols of c.right **do**
                Set c'.right to r
                if c' is not in C **then**
                    Insert c' in C
                    Insert c' in V
                    any_updates = true

        if d is Left **then**
            Insert c.right in R
            Set c'.right to the first $r-1$ symbols of c.right preceded by w
            Set c'.middle to the last symbol of c.left
            for each ngram $l \in L$ ending with the first $r-1$ symbols of c.left **do**
                Set c'.left to l
                if c' is not in C **then**
                    Insert c' in C
                    Insert c' in V
                    any_updates = true

    if not any_updates **then**
        return `NONHALT` ▷ Set C is closed, and does not include undefined transitions: the machine doesn't halt

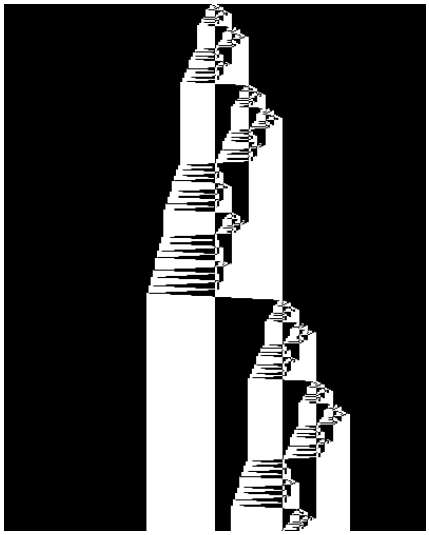


Figure 9: 10,000-step space-time diagram of a "fractal-looking" 5-state Turing machines that is solved by the LRU augmentation but has no known solution with standard NGramCPS or the fixed-length history augmentation, see Section 4.3.2. 1RB0RA_1LC---_1RC1LD_0LE1RA_0LC0LE

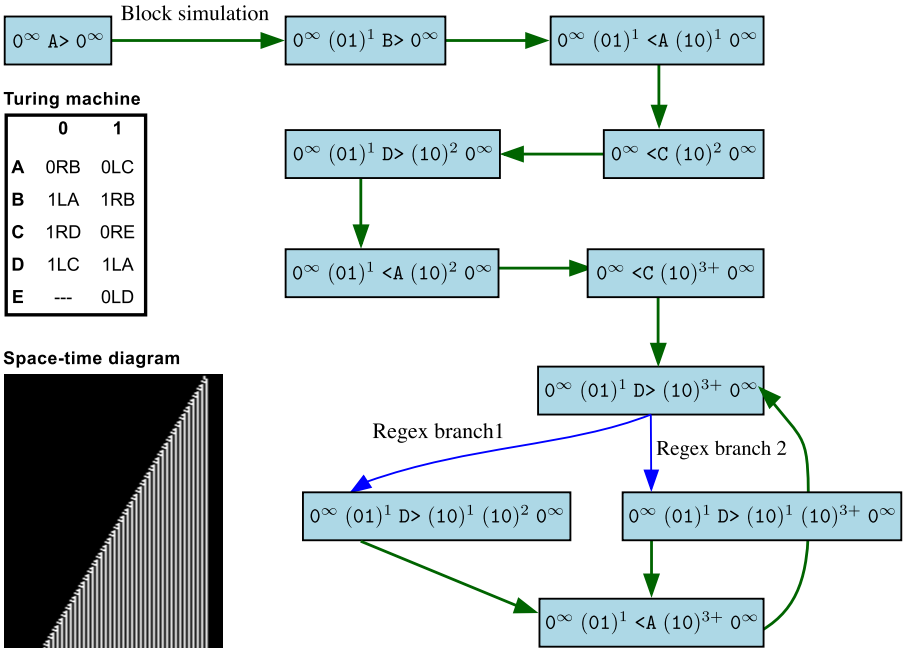


Figure 10: **RepWL graph.** Closed graph of regex configurations constructed by the Repeated Word List (RepWL) method (Section 4.4) for machine 0RB0LC_1LA1RB_1RD0RE_1LC1LA_---0LD with block size $l = 2$ and block repeat threshold $T = 3$. Block simulation and regex branching steps (see Section 4.4) are illustrated using respectively green and blue arrows. As illustrated by its 300-step space-time diagram, the machine is a simple Translated Cycler which can be easily handled by Section 1, but this machine has the pedagogical advantage of having a very small graph. Because the graph is closed and contains not halting configuration, the machine does not halt, Theorem 4.8.

4.4 Repeated Word List (RepWL)

4.4.1 Algorithm

The Repeated Word List (RepWL) technique, introduced by mxdys in Coq-BB5, is based on the following simple idea: if a word (or *block*) of length $l > 0$ appears consecutively on the tape more than $T > 0$ times (with $l, T \in \mathbb{N}$ fixed) then, we assume it may repeat an unbounded number of times in the future. In

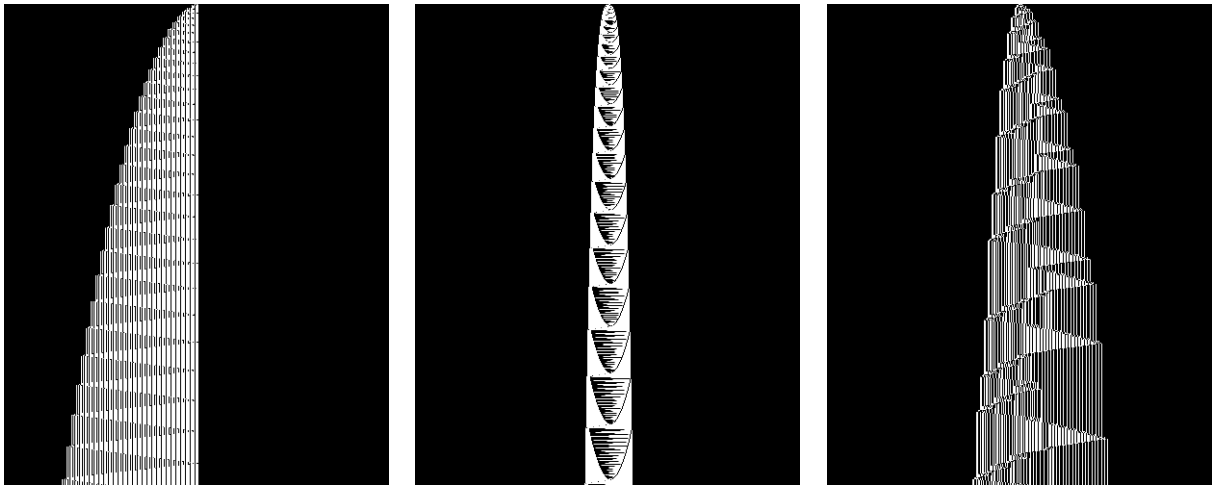


Figure 11: 10,000-step space-time diagrams of three 5-state machines decided by the Repeated Word List (RepWL) decider, Algorithm 3. Left: `1RBORD_0LC0LA_0LD1LC_1RA0LE_0RC---`. Center: `1RB---_1LB1RC_1RA1RD_1LE0RD_0LB0LC`. Right: `1RB---_0RC1RD_0LD1RC_1LE0RA_1RA0LE`. The RepWL graphs of these machines respectively have 42 and 845 and 143,181 nodes, ranging the entire distribution of RepWL node counts for 5-state machines, see Section 4.4.2. RepWL parameters (l, T) for these machines are respectively: $(5, 2)$, $(2, 3)$, and, $(20, 2)$.

practice, it means we represent configurations as regular expressions. For instance, consider the following configuration:

$$0^\infty\ 11100\ \texttt{A>}\ 11110101010111111111\ 0^\infty$$

Using $l = 2$ and $T = 3$, we represent it as:

$$0^\infty\ (01)\ (11)\ (00)\ \texttt{A>}\ (11)^2\ (01)^{3+}\ (11)^{3+}\ 0^\infty$$

Where blocks are constructed from the head outwards, and pumping symbols `0` out of 0^∞ if the number of symbols on either part of the tape is not a multiple of l. Any repetition of more than T times the same word $w \in \{0, 1\}^l$ is replaced by the regular expression $(w)^{T+}$ meaning that word w is repeated at least T times, hence the only exponents to ever be used in this representation are $\{1, 2, \ldots, T-1\}$ and $T+$. We call T the block-repeat threshold. Note that here, we use *directional head notation* for Turing machines, where the head lives in between cells and points either right or left. This framework is equivalent to the Turing machines setup used elsewhere in this work, see Section 2.

RepWL graph. Using the rules explained below (*block simulation* and *regex branching*), DECIDER-REPWL (Algorithm 3) simulates Turing machines directly on these regex configurations starting from the initial configuration (i.e. 0^∞ `A>` 0^∞), as to create a graph of such regex configurations to explore. If this graph is eventually closed (Algorithm 3 l.27) and contains no halting configuration then we know that the machine will never halt (Theorem 4.8) since we have constructed a set of configurations bigger than the one it will visit and that does not contain any halting transition. Because there is no guarantee the graph is closed, we also need an additional gas parameter (named N in Algorithm 3) indicating how many distinct nodes we're willing to visit at most. Figure 10 gives the RepWL graph of a simple machine.

For simulating Turing machines on regex configurations we need to deal with two cases: (i) *block simulation* when the head is facing a constant block (i.e. block without a +), such as `A>` $(11)^2$ and (ii) *regex branching* when the head is facing a block with a +, e.g. `D>` $(01)^{3+}$.

Block simulation. When the head is facing a constant block, such as in the above example `A>` $(11)^2$ (or if the head is facing 0^∞, we add constant block $(0^l)^1$), we can proceed to *block simulation*. Block simulation consists of simulating the Turing machine until the head eventually leaves the block or until a maximum step limit is reached (parameter named B in Algorithm 3). Note that the TM may never leave the block if it enters an infinite cycle which is why we need the step limit – one could alternatively implement cycle detection (Section 4.2) in block simulation but it is not the route taken in Coq-BB5.

Depending upon which Turing machine is being simulated, block simulation from block simulation from A> $(11)^2$ could produce, for instance, $(00)^2$ B> or <C 10 11 or enter a cycle and never leave the block. After block simulation, identical contiguous blocks are regrouped into powers, e.g. $(10)^1$ $(10)^1$ B> becomes $(10)^2$ B> and, assuming $T = 3$, $(10)^2$ $(10)^1$ B> would become $(10)^{3+}$ B>. In Figure 10, block simulation makes 0^∞ A> 0^∞, which is first internally replaced with 0^∞ A> $(00)^1$ 0^∞, go to 0^∞ $(01)^1$ B> 0^∞.

Regex branching. When the head is facing a block with a $^+$, for instance in Figure 10 we have 0^∞ 01^1 D> $(01)^{3+}$ 0^∞, from we add two configurations to the set of configurations to visit next:

1. **Regex branch 1.** We visit 0^∞ 01^1 D> $(01)^1$ $(01)^2$ 0^∞.
2. **Regex branch 2.** We visit 0^∞ 01^1 D> $(01)^1$ $(01)^{3+}$ 0^∞.

In both cases, we've reduced to block simulation.

Example 4.7. Figure 10 gives the RepWL graph for machine 0RB0LC_1LA1RB_1RD0RE_1LC1LA_---0LD. This machine was chosen to illustrate the method on a small RepWL graph but the machine is a simple Translated Cycler that be decided using Section 4.2.

Theorem 4.8 (Coq-BB5: `Lemma RepWL_ES_decider_spec`)**.** Let $\mathcal{M}$ be a Turing machine $l \in \mathbb{N}^+$ the block-length parameter, $T \in \mathbb{N}^+$ the block-repeat threshold, $B \in \mathbb{N}$ the maximum number of steps allowed in block simulation and $N \in \mathbb{N}$ the maximum number of nodes we are willing to visit. Then, DECIDER-REPWL($\mathcal{M}$, l, T, B, N) terminates and its result is correct – see Algorithm 3: if it returns NONHALT then $\mathcal{M}$ does not halt from the all-0 tape.

Proof. The algorithm terminates thanks to parameters B and N. For a machine $\mathcal{M}$, the algorithm returns NONHALT (Algorithm l.27) iff the RepWL graph of $\mathcal{M}$ contains less than N nodes (i.e. is closed), and contains no halting configuration (Algorithm l.19). Since the set of configurations reached by the machine is a subset of the regular language consisting of the union of each node's regex configuration, which includes no halting configuration, we get that the machine cannot halt from the all-0 tape. □

4.4.2 Implementations and results

Coq-BB5 implements RepWL (Algorithm 3), see function `RepWL_ES_decider`. Contrarily to previously presented deciders, RepWL is not applied in bulk using generic parameters. Instead, the 6,577 machines it decides are hardcoded in the proof together with the specific l and T parameters that decide them – see file `Decider_RepWL_Hardcoded_Parameters.v`. Figure **??** displays the (l, T) pairs used for these 6,577 machines. These parameters were found using a grid search in C++. For all these machines, maximum block simulation parameters and maximum number of graph nodes parameters are respectively set to 451 and 150,001.

Figure 11 gives space-time diagrams for machines with RepWL graphs with 42 (left), 845 (center) and 143,181 nodes. The machines on the left and on the right fit in the zoological category of *Bouncers*, see Section 7. We developed a dedicated decider for solving Bouncers, but it was not used in Coq-BB5 [62].

In the $S(4)$ pipeline (Table 5), RepWL is only used to decide two machines using parameters $l = 4$ and $T = 3$. These 4-state machines are given in Figure 12, and they respectively have 3,130 and 3,076 nodes in their RepWL graph.

Other implementations. At the time of this writing, RepWL also has a Haskell and a Python implementation [56, 66].

Algorithm 3 DECIDER-REPWL

Input: A Turing machine $\mathcal{M}$, block-length parameter $l > 0$, minimum size of nonconstant blocks $T > 0$, maximum number of steps allowed in block simulation $B \in \mathbb{N}$, maximum number of distinct nodes we're willing to visit $N \in \mathbb{N}$.
Output: `NONHALT` if the decider detects that the machine doesn't halt and `UNKNOWN` otherwise.

`to_visit` = [`A>`]
$V = \{\}$ ▷ Visited regex configurations

while $|V| < N$ **and** `to_visit.len()` $\neq 0$ **do**
    `regex_config` = `to_visit.`**pop**()

    if `regex_config` is in V **then**
        continue

    Insert `regex_config` in V

    if head is facing a constant block **then**
        `new_regex_config` = `regex_config.`**block_simulation**(B)
        if `new_regex_config` has halted (i.e. undefined transition was met) **or**
          limit B was exceeded during block simulation **then**
            return `UNKNOWN`
        `to_visit.`**append**(`new_regex_config`)
    else ▷ Head is facing a block with a +
        `regex_config_1`, `regex_config_2` = `regex_config.`**regex_branching**(M)
        `to_visit.`**append**(`regex_config_1`)
        `to_visit.`**append**(`regex_config_2`)

if $|V| < N$ **then**
    return `NONHALT`
else
    return `UNKNOWN`

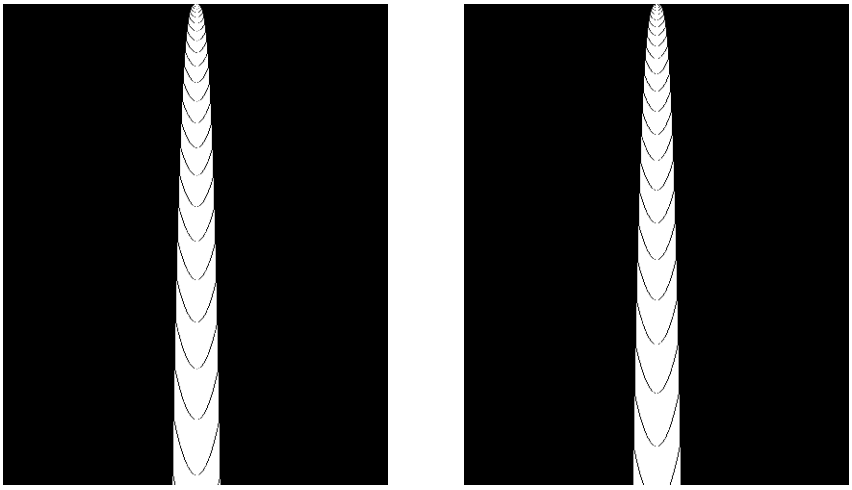
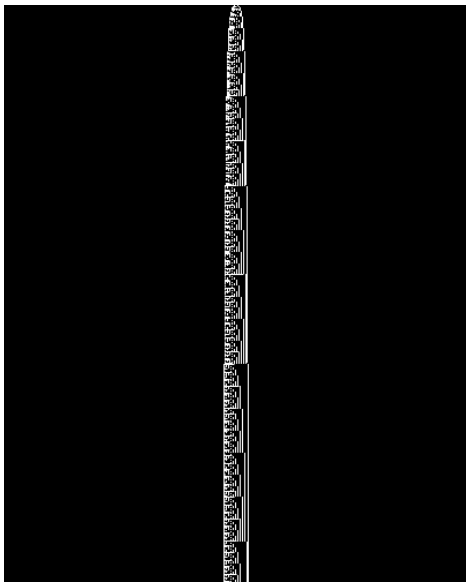


Figure 12: 10,000-step space-time diagrams of the two 4-state machines decided by the Repeated Word List (RepWL) decider in the $S(4)$ pipeline, Table 5. Left: `1RB1LA_1LA0RC_1LD1RC_---0LA`. Right: `1RB0RB_1LC1RB_---0LD_1RA1LD`. The RepWL graphs of these machines respectively have 3,130 and 3,076 nodes. Both machines are decided using $l = 4$ and $T = 3$.

4.5 Finite Automata Reduction (FAR)



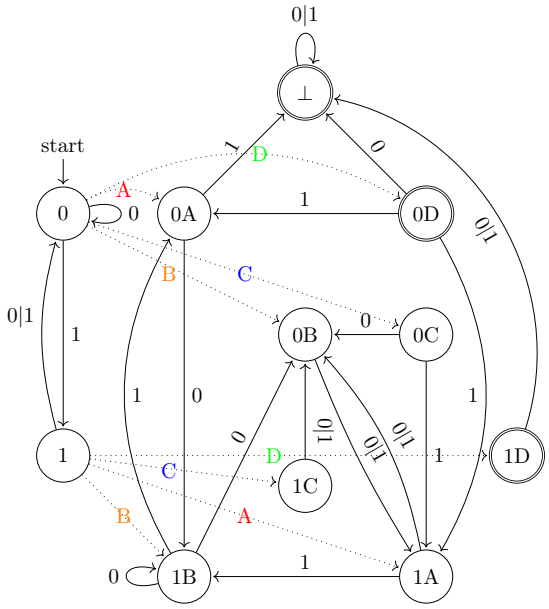


Figure 13: Left: 20,000-step space-time diagram of 4-state machine `1RB0LD_1LC1RA_0RB0LC_---1LA` – we use a 4-state machine to have a small FAR Nondeterministic Finite Automaton (NFA). Right: NFA that satisfies the FAR conditions (Theorem 4.9, with accepted steady state-set $\{\bot\}$) for this machine and hence is a certificate that the machine does not halt. This NFA accepts at least all eventually-halting configurations[31] of the machine (configurations are represented as words, see Section 4.5.2); because it rejects the initial all-0 configuration (e.g. word-encoded as `A0`, or just `A`, not leading to an accept state), we know the machine does not halt.

4.5.1 Overview

Finite Automata Reduction (FAR) is a *co-CTL* technique, i.e. it is dual to the Closed Tape Language (CTL) framework given in Section 4.1.2: for a given Turing machine, we are looking for a regular language that contains the set of the machine's eventually-halting configurations and, provided that the all-0 configuration is not in the regular language, we know that the machine does not halt.

The specificity of FAR is to restrict regular languages to a class of Nondeterministic Finite Automata (NFA) – those satisfying Theorem 4.9 – for which it is computationally simple to verify that they have the co-CTL properties: (i) reject the all-0 initial configuration, (ii) closed under Turing machines transitions, (iii) accept all eventually-halting configurations.

In Coq-BB5, FAR is only used as a **verifier** meaning that specific Turing machines together with their FAR NFAs are directly hardcoded in the proof (in file `Verifier_FAR_Hardcoded_Certificates.v`) and then verified using Theorem 4.9 – see Section 4.5.3 for results. FAR was originally developed by Blanchard as a fully-fledged decider – i.e. the verifier together with search algorithms for NFAs [4].

Here, we only present the verifier part of FAR (Theorem 4.9) while we present the decider and its variations in [62]. Figure 13 (right) gives a FAR NFA (i.e. satisfying[32] Theorem 4.9) for machine `1RB0LD_1LC1RA_0RB0LC_---1LA`: the NFA accepts at least all the eventually-halting configurations[31] of the machine and rejects the initial all-0 configuration (i.e. `A0` does not lead to an accept state), giving a certificate that the machine does not halt.

4.5.2 FAR theorem

In the following, we limit ourselves to Turing machines configurations with finite support, i.e. configurations with finitely many `1`s (or, more generally, finitely many non-`0` symbols) and, when we write *configuration*, we mean, *configuration with finite support*.

[31] With finitely many `1`s, see Section 4.5.2.
[32] Using accepted steady state-set $\{\bot\}$, see Section 4.5.2.

A Turing machine configuration c is represented as a finite word, called a *word-representation* of c, by concatenating the tape content (from left to right, making sure to include all the `1`s) and adding the state (in our case, a letter from A to E) just before the position of the head. For instance, two word-representations of the configuration $\texttt{0}^\infty$ `A>` `0011` $\texttt{0}^\infty$, are $\hat{c} =$ `A0011` and $\hat{c}' =$ `000A00110000`. Similarly, the initial all-0 configuration can be encoded as `A0` or even just `A`. Word-representations of the same configuration will only differ in the number of leading and trailing `0`s that they have.

Then, a co-CTL regular language of word-represented configurations $\mathcal{L}$ for a Turing machine $\mathcal{M}$ satisfies:

$$u \in \mathcal{L} \iff 0u \in \mathcal{L} \qquad \text{(leading zeros ignored)} \tag{4.1}$$

$$u \in \mathcal{L} \iff u0 \in \mathcal{L} \qquad \text{(trailing zeros ignored)} \tag{4.2}$$

$$c \to \bot \implies \hat{c} \in \mathcal{L} \qquad \text{(recognising halt, base case)}$$

$$(c_1 \to c_2) \wedge \hat{c}_2 \in \mathcal{L} \implies \hat{c}_1 \in \mathcal{L} \qquad \text{(recognising halt, induction)}$$

With c, c_1, c_2 configurations of $\mathcal{M}$ (with finite support) and $\hat{c}, \hat{c}_1, \hat{c}_2$ any of their word-representations. Given how word-representations are defined, the last two above conditions become:

$$\forall u, z \in \{\texttt{0},\texttt{1}\}^*\colon\ ufrz \in \mathcal{L}, \text{ if } \delta(f,r) \text{ is undefined (i.e. halting)} \tag{4.3}$$

$$\forall u, z \in \{\texttt{0},\texttt{1}\}^*, \forall b \in \{\texttt{0},\texttt{1}\}\colon utbwz \in \mathcal{L} \implies ubfrz \in \mathcal{L}, \text{ if } \delta(f,r) = (w, \mathrm{L}, t) \tag{4.4}$$

$$\forall u, z \in \{\texttt{0},\texttt{1}\}^*, \forall b \in \{\texttt{0},\texttt{1}\}\colon uwtz \in \mathcal{L} \implies ufrz \in \mathcal{L}, \text{ if } \delta(f,r) = (w, \mathrm{R}, t) \tag{4.5}$$

With $f, t \in \{A, B, C, D, E\}$ the "from" and "to" states in a transition, $r, w, b \in \{\texttt{0},\texttt{1}\}$ the bit "read", the bit "written", and just a bit, and δ the transition table (see Section 2) of $\mathcal{M}$.

We now transform Conditions (4.1)–(4.5) into, sometimes stronger, conditions on the structure of NFAs – using the usual linear-algebraic description of NFAs, which we first recall. Let $\mathbf{2}$ denote the Boolean semiring $\{0,1\}$ with operations $+$ and $\cdot$ respectively implemented by OR and AND [13]. Let $M_{m,n}$ be the set of matrices with m rows and n columns over $\mathbf{2}$. We may define a Nondeterministic Finite Automaton (NFA) with n states and alphabet $\mathcal{A}$ as a tuple $(q_0, \{T_\gamma\}_{\gamma\in\mathcal{A}}, a)$ where $q_0 \in M_{1,n}$ and $a \in M_{1,n}$ respectively represent the initial states and accepting states of the NFA. (i.e. if the i^{th} state of the NFA is an initial state then the i^{th} entry of q_0 is set to 1 and the rest are 0, and the i^{th} entry of a is set to 1 if and only if the i^{th} state of the NFA is accepting), and where transitions are matrices $T_\gamma \in M_{n,n}$ for each $\gamma \in \mathcal{A}$ (i.e. the entry (i,j) of matrix T_γ is set to 1 iff the NFA transitions from state i to state j when reading γ). Furthermore, for any word $u = \gamma_1 \dots \gamma_\ell \in \mathcal{A}^*$, let $T_u = T_{\gamma_1} T_{\gamma_2} \dots T_{\gamma_\ell}$ be the state transformation resulting from reading word u (Note: $T_\epsilon = I$). A word $u = \gamma_1 \dots \gamma_\ell \in \mathcal{A}^*$ is accepted by the NFA iff there exists a path from an initial state to an accepting state that is labelled by the symbols of u, which algebraically translates to $q_0 T_u a^T = 1$ with $a^T \in M_{n,1}$ the transposition of a.

Using this algebraic framework[33], Conditions (4.1) and (4.2) are implied by the following stronger conditions on transition matrix $T_0 \in M_{n,n}$:

$$q_0 T_0 = q_0 \tag{4.6}$$

$$T_0 a^T = a^T \tag{4.7}$$

Indeed, Condition (4.6) transparently ignores leading zeros, Condition (4.7) means that for all accepting states of the NFA, reading a `0` is possible and leads to an accepting state since $T_0 a^T$ describes the set of NFA states that reach the set of accepting states a after reading a `0`.

Then, Conditions 4.3–(4.5) algebraically translate to:

$$\forall u, z \in \{\texttt{0},\texttt{1}\}^*\colon\ q_0 T_u T_f T_r T_z a^T = 1, \text{ if } \delta(f,r) \text{ is undefined (i.e. halting)}$$

$$\forall u, z \in \{\texttt{0},\texttt{1}\}^*, \forall b \in \{\texttt{0},\texttt{1}\}\colon q_0 T_u T_t T_b T_w T_z a^T = 1 \implies q_0 T_u T_b T_f T_r T_z a^T = 1, \text{ if } \delta(f,r) = (w, \mathrm{L}, t)$$

$$\forall u, z \in \{\texttt{0},\texttt{1}\}^*, \forall b \in \{\texttt{0},\texttt{1}\}\colon q_0 T_u T_w T_t T_z a^T = 1 \implies q_0 T_u T_f T_r T_z a^T = 1, \text{ if } \delta(f,r) = (w, \mathrm{R}, t)$$

These conditions are unwieldy. We seek stronger (thus still sufficient) conditions which are simpler:

- For machine transitions going left/right, simply require $T_t T_b T_w \preceq T_b T_f T_r$ and $T_w T_t \preceq T_f T_r$, respectively with $\preceq$ the following relation on same-size matrices: $M \preceq M'$ if $M_{ij} \leq M'_{ij}$ element-wise, that is, if the second matrix has at least the same 1-entries as the first matrix.

[33] In the following, we limit ourselves to the binary tape alphabet $\{\texttt{0},\texttt{1}\}$, but the results generalise transparently to arbitrary alphabets $\mathcal{A}$.

- To simplify the condition for halting machine transitions: define an *accepted steady state-set* s to be a row vector such that $sa^T = 1$, $sT_0 \succeq s$, and $sT_1 \succeq s$. Given such s, we have that: $\forall q \in M_{1,n}\ q \succeq s \implies \forall z \in \{0,1\}^* : qT_z a^T = 1$. Assuming that such s exists we can simply require: $\forall u \in \{0,1\}^* : q_0 T_u T_f T_r \succeq s$ which is stronger than $\forall u, z \in \{0,1\}^* :\ q_0 T_u T_f T_r T_z a^T = 1$ with $\delta(f,r)$ an undefined transition.

Combining the above, we get FAR:

Theorem 4.9 (Coq-BB5: `Lemma dfa_nfa_verifier_spec`[34]**).** Machine $\mathcal{M}$, with transition table δ (see Section 2), doesn't halt from the initial all-0 configuration if there is a Nondeterministic Finite Automaton $(q_0, \{T_\gamma\}, a)$ and row vector s satisfying the below:

$$q_0 T_0 = q_0 \qquad \text{(leading zeros ignored)} \tag{4.6}$$
$$T_0 a^T = a^T \qquad \text{(trailing zeros ignored)} \tag{4.7}$$
$$sa^T = 1 \qquad (s \text{ is accepted}) \tag{4.8}$$
$$sT_0, sT_1 \succeq s \qquad (s \text{ is a steady state}) \tag{4.9}$$
$$\forall u \in \{0,1\}^* : q_0 T_u T_f T_r \succeq s \qquad \text{if } \delta(f,r) \text{ is undefined (i.e. halting)} \tag{4.10}$$
$$\forall b \in \{0,1\} : T_b T_f T_r \succeq T_t T_b T_w \qquad \text{if } \delta(f,r) = (w, \mathrm{L}, t) \tag{4.11}$$
$$T_f T_r \succeq T_w T_t \qquad \text{if } \delta(f,r) = (w, \mathrm{R}, t) \tag{4.12}$$
$$q_0 T_A a^T = 0 \qquad \text{(initial configuration rejected)} \tag{4.13}$$

Proof. Conditions (4.6)–(4.12) ensure that the NFA's language includes at least all eventually halting configurations of $\mathcal{M}$, see above. Condition (4.13) ensures that the initial all-0 configuration of the machine is rejected, hence not eventually halting. Hence, if conditions (4.6)–(4.13) are satisfied, we can conclude that $\mathcal{M}$ does not halt from the initial all-0 configuration. □

Verifier. Theorem 4.9 has the nice property of being easy to verify: given a Turing machine, an NFA and a vector s, the task of verifying that equations (4.6)–(4.13) hold and thus that the machine does not halt, is computationally simple[35]. For instance, it is easy to check that the NFA given in Figure 13 satisfies Theorem 4.9, using accepted steady state-set $\{\bot\}$, for machine `1RB0LD_1LC1RA_0RB0LC_---1LA` and hence, the NFA provides a certificate that the machine does not halt.

4.5.3 Implementations and results

Coq-BB5 implements Theorem 4.9 in the special case where the FAR NFA is computed from a *precursor* Deterministic Finite State Automaton, as described in [62] ("direct FAR algorithm"). Certificates, consisting of such DFAs are hardcoded in the proof for 23 machines (in file `Verifier_FAR_Hardcoded_Certificates.v`) and then verified using `dfa_nfa_verifier` (see file `Verifier_FAR.v`).

These certificates were either found using extensive compute (e.g. several weeks of searching DFAs essentially by bruteforce) or translated from other, undocumented, regular co-CTL methods (see Section 4.1.2); indeed, in [62] we show that that FAR is an *universal* regular co-CTL method: any regular co-CTL proof can be shoehorned in the framework of Theorem 4.9.

Other implementations. FAR has several other implementations:

1. Justin Blanchard's original, optimised, Rust implementation [4]
2. Tony Guilfoyle's C++ reproduction [63]
3. Tristan Stérin's Python reproduction [65]

[34] Coq-BB5's lemma is slightly different, as it builds the NFA satisfying this theorem using a given "precursor" Deterministic Finite Automaton (DFA) – as initially developed in [4] – see Section 4.5.3.

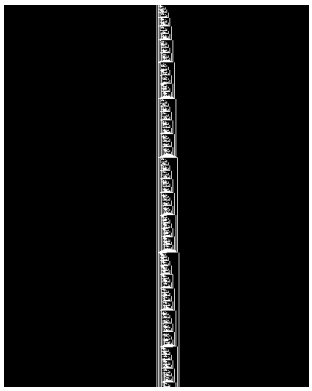
[35] Note that although equation (4.10) has a $\forall u \in \{0,1\}^*$ quantification, the set of NFA states reachable after reading an arbitrary $u \in \{0,1\}^*$ is computable, and we just have to consider one instance of equation (4.10) replacing $q_0 T_u$ per such state.

4.6 Weighted FAR (WFAR)

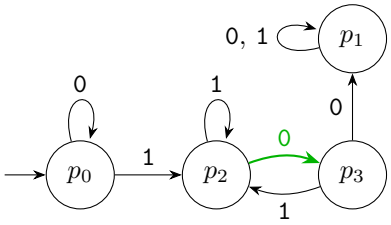
(a) Turing machine

	0	1
A	1RB	—
B	0RC	1LC
C	1RD	1RC
D	1LE	1LD
E	0RA	0LE

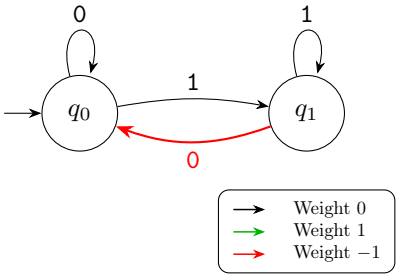
(a') Space-time diagram



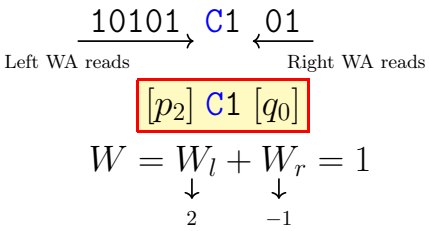
(b) Left Weighted Automaton



(c) Right Weighted Automaton



(d) Example: configuration is accepted, hence nonhalting



Configuration accepted, see (e), hence machine does not halt starting from 10101 C1 01, see Theorem 4.10.

(e) Accepted weighted configurations

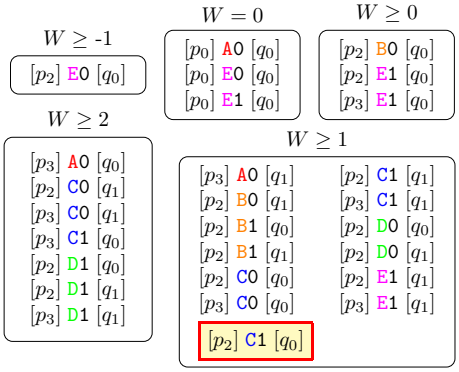


Figure 14: WFAR certificate of nonhalting for machine 1RB---_ORC1LC_1RD1RC_1LE1LD_ORA0LE: (a) transition table and 20,000-step space-time diagram, (b) left weighted automaton: processes symbols to the left of the head in the left-to-right direction, which results in a left end-state – e.g. state p_2 when processing 10101 – and a left weight obtained by summing the weights of each transition – e.g. $W_l = 2$ when processing 10101 (c) right weighted automaton: processes symbols to the right of the head (excluding the symbol read by the head) in the right-to-left direction – indicated with arrow –, which results in a right end-state – e.g. state q_0 when processing 01 right-to-left – and a right weight – e.g. $W_r = -1$ when processing 01 right-to-left (d) example, the total weight of configuration 10101 C1 01 is $W = W_l + W_r = 1$, using same word-encoding of configurations as in Section 4.5, and the right and left end-states are p_2 and q_0. Weighted automaton configuration $[p_2]$ C1 $[q_0]$ with $W = 1$ is in the set of accepted weighted configurations (under more general $W \geq 1$), see (e). Therefore we know that the machine does not halt from configuration 10101 C1 01, Theorem 4.10. Similarly, Turing machine configuration A0, which results in weighted configuration $[p_0]$ A0 $[q_0]$ with $W = 0$ is accepted, ensuring that the machine does not halt from the all-0 initial tape, Theorem 4.10.

4.6.1 Overview

Weighted automata are an extension of usual finite state automata where each transition is given a weight in $\mathbb{Z}$: when a word is processed, total weight $W \in \mathbb{Z}$ is computed by summing the weights of all the encountered transitions. Accepted words are described by a set of pairs of final-state and weigth lower and upper bounds (potentially infinite) to satisfy: for instance, the archetypal nonregular language 0^n1^n

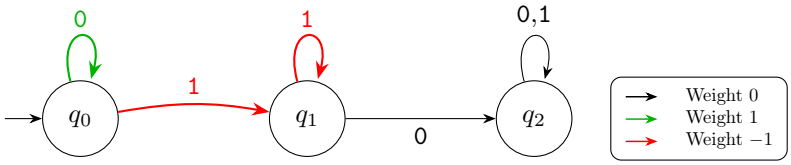


Figure 15: Weighted automaton recognising nonregular language $\texttt{0}^n\texttt{1}^n$, using accept set $\{(q_1, W = 0)\}$ or $\{(q_1, W = 0), (q_0, W = 0)\}$ if we include the empty word.

is recognised by the weighted automaton of Figure 15 using accept set $\{(q_1, 0 \leq W \leq 0)\}$ which we can simplify as $\{(q_1, W = 0)\}$ and we may add $(q_0, W = 0)$ to the set if we want to include the empty word.

Weighted Finite Automata Reduction (WFAR) is an extension of FAR (Section 4.5) using deterministic weighted finite automata. Figure 14 gives a *WFAR automaton*, which is a certificate of nonhalting the machine given in Figure 14 (a). A WFAR automaton consists of (i) a *left deterministic weighted automaton* (ii) a *right deterministic weighted automaton* and (iii) a set of accepted *weighted configurations*, see Figure 14 (b), (c), and (e). A WFAR automaton processes word-representations (as defined in Section 4.5) of Turing machine configurations[36] in the way describe below, and, if a configuration is accepted by the WFAR automaton, we know that the associated Turing machine does not halt from that configuration, Theorem 4.10. That way, WFAR is a CTL method (instead of co-CTL for FAR), see Section 4.1.2. The WFAR automaton of Figure 14 accepts (see below for what it means) the initial configuration `A0`, giving a certificate of nonhalting for the machine of Figure 14 (a) from the all-0 tape.

The method was initially developed by Iijil as a decider [29] and integrated to Coq-BB5 by mxdys as a verifier: similarly to FAR (Section 4.5), 17 WFAR certificates are directly hardcoded in the Coq proof, see file `Verifier_WFAR_Hardcoded_Certificates.v`, see Section 4.6.3 for results.

WFAR processing. Let's describe how a WFAR automaton processes a word-represented Turing machine configuration in order to decider whether it is accepted or not, as illustrated in Figure 14. WFAR is an extension of the "Meet-in-the-middle"[37] instance of FAR [62]: word-representations of configurations are split into three segments, (i) word to the left of the head, (ii) head state and symbol, (iii) word to the right of the head; e.g. `10101 C1 01`, Figure 14 (d). The left word – here `10101` – is processed left-to-right by the left weighted automaton, Figure 14 (b), and the right word – here `01` – is processed right-to-left, by the right weighted automaton, Figure 14 (c). In this case, this results in final left state p_2, final right state q_0, final left weight $W_l = 2$ and final right weight $W_r = -1$; the final total weight is $W = W_l + W_r = 1$, Figure 14 (d). We denote this final *weighted configuration* as $[p_2]$ `C1` $[q_0]$ with $W = 1$. This final weighted configuration belongs to the set of accepted weighted configurations, Figure 14 (e), which means that configuration `10101 C1 01` is *accepted* by this WFAR automaton.

4.6.2 WFAR theorem

WFAR is CTL technique – see Section 4.1.2: a WFAR automaton for a given Turing machine is meant to recognise a language of configurations $\mathcal{L}$ that includes the initial all-0 configuration, closed under Turing machine steps and that does not contain any halting configuration. Hence, we get the following CTL formalism, plus leading/trailing zeros conditions similarly to FAR:

$$u \in \mathcal{L} \iff 0u \in \mathcal{L} \qquad \text{(leading zeros ignored)} \qquad (4.1)$$

$$u \in \mathcal{L} \iff u0 \in \mathcal{L} \qquad \text{(trailing zeros ignored)} \qquad (4.2)$$

$$c \to \bot \implies \hat{c} \notin \mathcal{L} \qquad \text{(reject halt)} \qquad (4.14)$$

$$(c_1 \to c_2) \wedge \hat{c}_1 \in \mathcal{L} \implies \hat{c}_2 \in \mathcal{L} \qquad \text{(forward closure)} \qquad (4.15)$$

Let's now show how to verify that a given WFAR automaton for a given Turing machine $\mathcal{M}$ accepts such $\mathcal{L}$, hence providing a certificate that $\mathcal{M}$ does not halt from the all-0 tape.

In the following, $\delta_L : Q_L \times \{0, 1\} \to Q_L$ and $\delta_R : Q_R \times \{0, 1\} \to Q_R$ respectively refer to the transition functions of the deterministic left and right weighted automaton of a WFAR automaton, e.g. Figure 14 (b) and (c), with $Q_L = \{p_0, \ldots, p_{n_L-1}\}$ and $Q_R = \{q_0, \ldots, q_{n_R-1}\}$ their respective set of states with n_L and n_R the number of left/right states and p_0 and q_0 are the respective initial states of the left and right

[36] With finitely many 1s, which we always assume from now on.
[37] See Section 6.6 in [62].

weighted automaton. Weights are given by $w_L : Q_L \times \{0,1\} \to \mathbb{Z}$ and $w_R : Q_R \times \{0,1\} \to \mathbb{Z}$. We write $\delta_{\mathcal{M}} : \mathcal{S} \times \{0,1\} \hookrightarrow \{0,1\} \times \{\mathrm{L},\mathrm{R}\} \times \mathcal{S}$ for the transition function of $\mathcal{M}$[38].

Leading/trailing zeros. Checking Conditions (4.1) and (4.2) for a WFAR automaton is simple: thanks to the left-to-right and right-to-left respective read directions for the left and right weighted automaton, we simply have to check that $\delta_L(p_0, 0) = p_0$ and $\delta_R(q_0, 0) = q_0$ as well as $w_L(p_0, 0) = w_R(q_0, 0) = 0$ to ensure the convention that the weight of all word-representations of the initial all-0 configuration is 0.

Forward closure, without weights: back to FAR. First, let's reformulate forward closure for a WFAR automaton, ignoring weights computations. Forward closure concerns the WFAR automaton's accept state, let's consider an example first. The WFAR automaton of Figure 14 accepts the initial Turing machine configuration A0: the WFAR configuration $[p_0]$ A0 $[q_0]$ (ignoring $W = 0$) is in the accept set given in Figure 14 (e). To ensure forward closure, Condition 4.15, let's consider how $\delta_{\mathcal{M}}(\mathtt{A}, 0) = 1\mathrm{R}\mathtt{B}$ affects $[p_0]$ A0 $[q_0]$; we get $[p_0]$ 1 B? [?], which is $[p_2]$ B? [?] given that $\delta_L(p_0, 1) = p_2$, see Figure 14 (b). In order to resolve ?, we look at all the transitions in the right weighted automaton that lead to q_0, see Figure 14 (c): there are two, both reading a 0, giving $[p_2]$ B0 $[q_0]$ and $[p_2]$ B0 $[q_1]$. Ignoring weights, we want both in the accept set:[39] that ensures that for any Turing machine configuration c_1 yielding WFAR configuration $[p_0]$ A0 $[q_0]$, then c_2 is also accepted by the WFAR automaton with $c_1 \to c_2$. Note that c_1 and c_2 are not necessarily reachable from the initial all-0 tape: CTL methods provide languages that over-estimate the language generated by Turing machines from the all-0 tape.

In general, ignoring weights, forward closure means the following for WFAR automaton accept set $\mathfrak{A}$:

$$\forall q', r' \in Q_R \times \{0,1\} \text{ s.t. } \delta_R(q', r') = q, \quad [p]\, fr\, [q] \in \mathfrak{A} \Rightarrow [\delta_L(p, b)]\, tr'\, [q'] \in \mathfrak{A} \quad \text{if } \delta_{\mathcal{M}}(f, r) = (b, \mathrm{R}, t) \tag{4.16}$$

$$\forall p', r' \in Q_L \times \{0,1\} \text{ s.t. } \delta_L(p', r') = p, \quad [p]\, fr\, [q] \in \mathfrak{A} \Rightarrow [p']\, tr'\, [\delta_R(q, b)] \in \mathfrak{A} \quad \text{if } \delta_{\mathcal{M}}(f, r) = (b, \mathrm{L}, t) \tag{4.17}$$

For all left/right weighted automata states $p, q \in Q_L \times Q_R$ and notations $f, t \in \{A, B, C, D, E\}$ the "from" and "to" states in a transition, $r, b \in \{0,1\}$ respectively the bit read and the bit written in a transition. If, ignoring weights, a WFAR accept set is forward-closed in the above sense, contains no halting configuration, and contains $[p_0]$ A0 $[q_0]$, then we are in a particular case of FAR, as shown in [62] i.e. Theorem 4.9 can be applied: the Turing machine does not halt from the initial all-0 configuration and, is regular in the sense of Section 4.1.2.

Forward closure, with weights: beyond FAR. Weights allow to further restrict the accept set in cases where the above, *weight-less*, forward closure does include halting configurations. For instance, in the case of Figure 14, we have $[p_2]$ D1 $[q_1]$ (with $W \geq 2$) in the accept set $\mathfrak{A}$, given in Figure 14 (e), and, computing weight-less forward closure from this WFAR configuration, ignoring weights, yields, using (4.16) and, (4.17), $[p_0]$ D1 $[q_1]$, then $[p_0]$ D0 $[q_1]$, then $[p_0]$ E0 $[q_1]$ and finally, $[p_0]$ A1 $[q_0]$, which is a halting configuration, meaning that we cannot conclude that $\mathcal{M}$ does not halt from the initial all-0 configuration. However, looking at Figure 14 (e) we see that $[p_0]$ D1 $[q_1] \notin \mathfrak{A}$ and hence none of the successors either. This *refinement* of $\mathfrak{A}$ is due to discarding *impossible* weighted configurations, which we explain now.

With weights, WFAR configurations are expressed as follows: $[p]\ fr\ [q];\ W \geq m;\ W \leq M$; with $m \in \mathbb{Z} \cup \{-\infty\}$ and $M \in \mathbb{Z} \cup \{+\infty\}$ weights bounds. For instance, considering the accept state $\mathfrak{A}$ of Figure 14 (e) with weights, we have that the initial weighted configuration, $c_1' = [p_0]$ A0 $[q_0];\ W \geq 0;\ W \leq 0$; is in $\mathfrak{A}$. When computing closure, bounds are updated by the *total weight change* incurred when processing weighted transitions: consider $c_2' = [p_2]$ B0 $[q_1];\ W \geq$?; $W \leq$? obtained from the initial weighted configuration by (4.16); we have $W(c_2) = W(c_1) + w_L(p_0, 1) - w_R(q_1, 0)$ with $c_1 \to c_2$ Turing machine configurations such that c_1 yields WFAR configuration c_1' and c_2 yields c_2'. Hence, for c_2' to be accepted, we must update its weight bounds by weight change $w_L(p_0, 1) - w_R(q_1, 0) = 0 - (-1) = +1$, giving $c_2' = [p_2]$ B0 $[q_1];\ W \geq 1;\ W \leq 1$, which is implied by more general $c_2' = [p_2]$ B0 $[q_1];\ W \geq 1;\ W < +\infty$ in $\mathfrak{A}$ of Figure 14 (e). In general, in the case of (4.16), using same notations, weights bounds m and M are added to weight change $w_L(p, b) - w_R(q', r')$ and weight change $w_L(p', r') - w_R(q, b)$ in the case of (4.17); infinite bounds remain the same under any weight change.

[38] In the following, we limit ourselves to the binary tape alphabet $\{0,1\}$, but the results generalise transparently to arbitrary alphabets $\mathcal{A}$.

[39] Which is the case here, with $W \geq 0$ and $W \geq 1$ in Figure 14 (e).

Coming back to $[p_2]$ D1 $[q_1]$; $W \geq 2$; $W \leq +\infty$; which we have shown above to lead to a halting configuration, we can now compute the bounds of the weighted configuration we obtained using (4.17): $[p_0]$ D1 $[q_1]$ $W \geq 2$; $W < +\infty$; as there is no weight changes. However, note that any left word reaching q_0 has left weight $W_l = 0$ and any right word reaching q_1 has right weight $W_r \leq 0$, hence total weight $W \leq 0$, which is incompatible with the constraint $W \geq 2$; hence we can discard $[p_0]$ D1 $[q_1]$ $W \geq 2$; $W < +\infty$ from the accept set $\mathfrak{A}$ as it is an impossible weighted configuration. Doing this also discards from $\mathfrak{A}$ all the weighted configurations we computed from $[p_0]$ D1 $[q_1]$ $W \geq 2$; $W < +\infty$, including the halting one, $[p_0]$ A1 $[q_0]$.

In this case, in order to conclude, we needed to know that, in the left weighted automaton of Figure 14 (b), terminating at state q_0 implies $W_l = 0$. In general, the exact feasible weight bounds of any state in a weighted automaton can be computed using the Bellman-Ford algorithm[40]: for instance, in the left weighted automaton of Figure 14 (b), at state p_2, we have $0 \leq W_l < +\infty$. Hence we can use these feasible weight bounds to automatically discard impossible weighted configurations from $\mathfrak{A}$ and hopefully, end up with no halting configuration in $\mathfrak{A}$.

We say that $\mathfrak{A}$ is *weighted forward closed* for machine $\mathcal{M}$ if (i) for all weighted configuration $c \in \mathfrak{A}$, for any weighted configurations c' obtained by closure using (4.17) or (4.16) together with the weights bounds update rules stated above, there is $c'' \in \mathfrak{A}$ with bounds m'' and M'' such that $m'' \leq m'$ and $M'' \geq M'$ with m' and M' the bounds of c' and (ii) there is no impossible weighted configuration in $\mathcal{A}$ as defined above, i.e. incompatible with the feasible weight bounds computed from the left and right weighted automata.

TS: Iijil commented that this definition needs improvement.

We finally get the WFAR theorem:

Theorem 4.10 (Coq-BB5: `Lemma MITM_WDFA_verifier_spec`)**.** Let $\mathcal{M}$ be a Turing machines and $\mathcal{W}$ be a WFAR automaton with accept set $\mathfrak{A}$ such that:

1. Leading and trailing zeros are ignored: $\delta_L(p_0, 0) = p_0$ and $\delta_R(q_0, 0) = q_0$ with $w_L(p_0, 0) = w_R(q_0, 0) = 0$ with δ_L and δ_R the transition functions of the left and right weighted automata of $\mathcal{W}$ and w_L and w_R their weighing functions.
2. The initial configuration is accepted: i.e. $[p_0]$ A0 $[q_0]$; $W = 0$; is in $\mathfrak{A}$.
3. $\mathfrak{A}$ is weighted forward closed for $\mathcal{M}$.
4. $\mathfrak{A}$ contains no halting configurations.

Then $\mathcal{M}$ does not halt for any configuration accepted by $\mathcal{W}$, which includes the initial all-0 configuration.

Proof. Point 1 guarantees that all the word-representations (see Section 4.5) of the same Turing machine configurations result in the same weighted WFAR configuration when processed by $\mathcal{W}$ (see Section 4.6.1). Points 2-4 are the WFAR reformulations of the CTL argument (Section 4.1.2). Hence, using the CTL argument, any Turing machine configuration accepted by $\mathcal{W}$ is nonhalting, and, in particular, the initial all-0 configuration. □

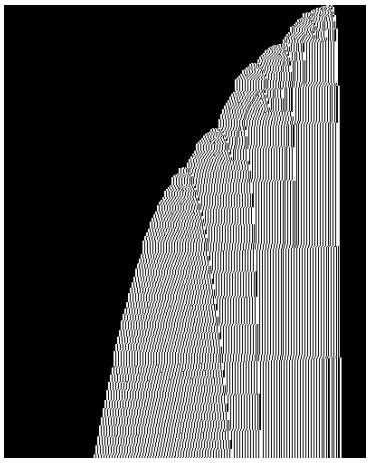
4.6.3 Implementations and results

Coq-BB5 implements Theorem 4.10, see file `Verifier_WFAR.v`. Certificates consist of left and right weighted automaton: accept sets are constructed by the Coq verifier which computes the closure from $[p_0]$ A0 $[q_0]$; $W = 0$ and uses an integer parameter P given in the certificate such that a bound $W \geq P$ is replaced by $W \leq +\infty$. See the 17 certificates in `Verifier_WFAR_Hardcoded_Certificates.v`.

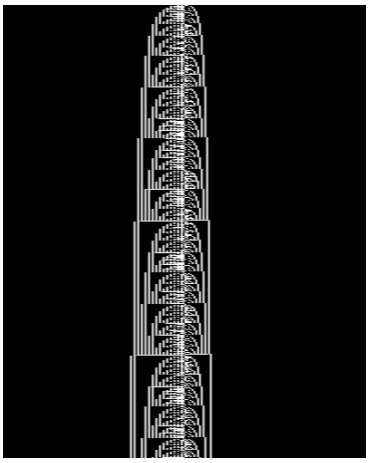
These certificates were mainly found by the original WFAR decider implementation [29] which searches the space of WFAs using bruteforce. Certificates for "Helices" (see Section 4.6) were handcrafted by Blanchard and are significantly bigger than the other certificates: about 50 states in the left and right WAs each where other certificates have less than 10 in each.

[40] Using Bellman-Ford was suggested by a LLM. However both the original and the Coq-BB5 implementations do not need it as they use restricted weighted automata on which it is easy to check whether the feasible weights for each state are nonpositive or nonnegative [29].

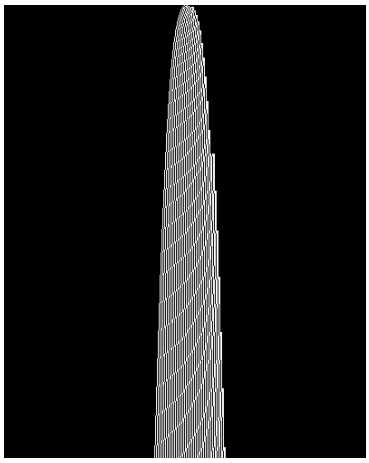
5 5-state Sporadic Machines



Skelet #1

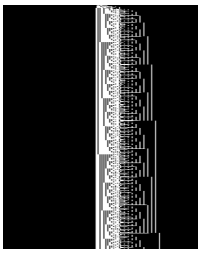


Skelet #10
Double Fibonacci Counter

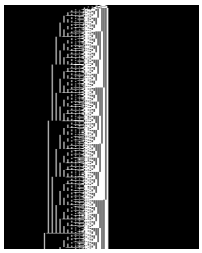


Skelet #17

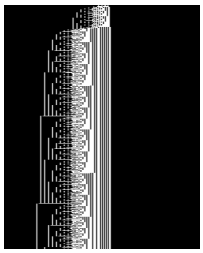
Shift Overflow Counters



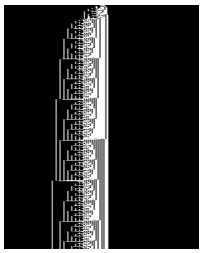
Skelet #15



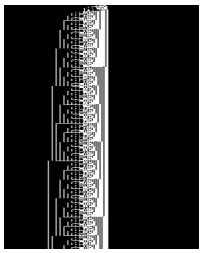
Skelet #26



Skelet #33

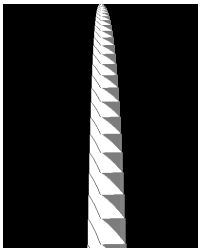


Skelet #34

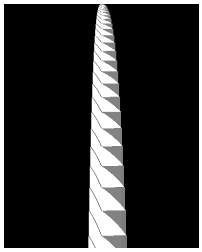


Skelet #35

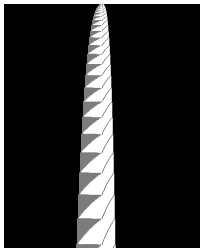
Finned Machines



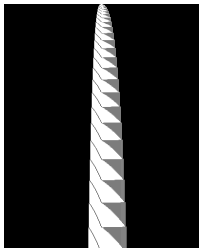
Finned #1



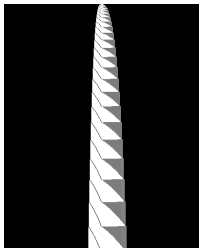
Finned #2



Finned #3



Finned #4



Finned #5

Figure 16: Family picture of the 5-state Sporadic Machines (20,000-step space-time diagrams) which required individual Coq nonhalting proofs; machine names in the Figure are clickable URLs giving the TNF-normalised transition table of each machine (see Section 3). All Sporadic Machines were also identified by Skelet [20] as holdouts of his `bbfind` program. For better visibility, diagrams of counters (Skelet #10 and Shift Overflow Counters) have been represented using a tape of length 200 instead of 400, giving a *zoomed-in* effect.

Sporadic Machines are 13 nonhalting 5-state Turing machines that were not captured by deciders (Section 4) and required individual Coq proofs of nonhalting; their space-time diagrams are given in Figure 16 where each name is a clickable URL leading to the machine's transition table and space-time diagram. Twelve of these machines, i.e. all but "Skelet #17" (see below), were proved nonhalting in `busycoq` [43], and then integrated[41] into Coq-BB5. Machine "Skelet #17" was the last 5-state machine to

[41] For convenience, the relevant parts of `busycoq` have been added to the root of Coq-BB5. Coq-BB5 translates `busycoq`

be formally proven nonhalting in Coq, as part of Coq-BB5, achieving the proof of $S(5) = 47{,}176{,}870$ – a different proof also had been released as a standalone paper, [72].

Interestingly all Sporadic Machines had been identified by Georgi Georgiev (also known as "Skelet", see Section 1.1) in 2003: either as part of his 43 unsolved machines[42] which are named after him, e.g. "Skelet #1" (see Figure 16), or, in the case of what we call "Finned Machines", marked by him as "easily provable by hand" [19]. Sporadic Machines can be arranged in three buckets:

- **Finned Machines.** These are five similar machines that hold 3 unary numbers on the tape (and can merge the middle one into its neighbor), vary them while maintaining a linear relation, and in the process ensure any deviation from this linear relation would be detected and cause a halt. These machines were solved by handcrafting nonhalting certificates similar in flavor to WFAR certificates (Section 4.6). The certificates were crafted by Blanchard, translated in Coq by mei, see `busycoq` files `Finned{1-5}.v`. An argument of irregularity (Section 4.1.2) was given for machine "Finned #3" [28]. A later-developed irregular extension of RepWL (Section 4.4) has been reported to solve these machines[43].

- **Shift Overflow Counters.** This family concerns Skelet's machines 15, 26, 33, 34 and 35; Figure 16. These machines are similar: they implement two independent binary counters, one to the left of the tape and the other to the right. Their behavior can be described by two distinct phases: an orderly "Counter Phase" where each counter is simply incremented and a more complex "Reset Phase" triggered by one of the counters overflowing. If the counter were to overflow again during a "Reset Phase", the machines would halt. Therefore, the proof of nonhalting depends upon demonstrating that the machine maintains a "Reset Invariant" throughout the "Reset Phase" which does not allow another overflow. These machines were first analysed and described by Ligocki, who provided an informal proof of "Skelet #34" as well as conjectures about the others [37]. They were then proved in Coq by Yuen and mei as part of `busycoq`, see files `Skelet{15,26,33,34,35}.v` [43].

- **Skelet #1, Skelet #10, and, Skelet #17.** These three machines each have unique behaviors which we detail below.

Skelet #1. This machine is a Translated Cycler, i.e. a machine that eventually repeats the same pattern translated in space (see Section 4.2), but with enormous parameters: its pre-period (number of steps to wait before the pattern first appears) is about 5.42×10^{51} and its period (number of steps taken by the repeated pattern) is $8,468,569,863$. This was discovered by means of accelerated simulation by Kropitz and Ligocki [51] and thorough analysis by Ligocki [39, 38]. The result was confirmed correct after mei formalised it in Coq as part of `busycoq`, see file `Skelet1.v` [43]. The 10^{51} pre-period was computed later by Huang [26].

Skelet #10 (Double Fibonacci Counter). This machine implements two independent *base Fibonacci* counters, one to the left of the tape and the other to the right. Counting in base Fibonacci means exploting Zeckendorf's theorem [70]: any natural number can be expressed as a sum of Fibonacci numbers in exactly one way, excluding using numbers immediately adjacent in the Fibonacci sequence, where the Fibonacci sequence is $F = 1, 2, 3, 5, 8, 13, 21, 34 \ldots$ – each number in the sequence is the sum of the two previous ones. For instance, $17 = 1 + 3 + 13$ and this decomposition would be represented as `100101` in big-endian binary: the i^{th} bit from the right is 1 if we use F_i in the sum. Each of the two counters of Skelet #10 enumerate natural numbers in base Fibonacci, using slightly different encodings and the machine halts iff the counters ever get out of sync – which, does not happen. The machine was analysed independently by Briggs and Ligocki [10, 40] and Ligocki's proof [40] was formalised in Coq by mei as part of `busycoq`, see file `Skelet10.v` [43]. Skelet #10 is the only known 5-state *double* Fibonacci counter, but there are several known *single* Fibonacci counters, such as `1RB0RA_0LC1RA_1LD0LC_1RE1LC_---0RB`, solved by Coq-BB5's NGramCPS (Section 4.3).

Skelet #17. *The final boss.* This machine manages a list of integers $n_1, \ldots, n_k \in \mathbb{N}$ represented in unary on the tape using encoding: $(10)^{n_1}1(10)^{n_2}1 \ldots 1(10)^{n_k}$. The list can only increase and undergoes a set of complex transformations related to Gray code, and, the machine halts iff $n_1 = n_2 = 0$ and $n_3, \ldots, n_k$ are all even, which, never happens. This description was first drafted by savask [55], proven in a standalone paper by Xu [72] and, finally, formalised in Coq by mxdys as part of Coq-BB5. Skelet #17 was the last 5-state machine to be solved.

proofs using `BusyCoq_Translation.v`.

[42] Apart from [19], these 43 machines are also listed here: `https://bbchallenge.org/skelet`.

[43] `https://discuss.bbchallenge.org/t/bb5s-finned-machines-summary/234`

6 Results

Coq-BB5 is available at https://github.com/ccz181078/Coq-BB5 and contains extensive instructions for how to compile the proof. The proof compiles in about 45 minutes using 13 cores on a standard laptop. The proof only relies on Coq's standard library axiom `functional_extensionality_dep`[44], which claims that two functions are equal if they are equal in all points.

Theorem 1.1 (Coq-BB5: `Lemma BB5_value`)**.** $S(5) = 47{,}176{,}870$.

Proof. The Coq proof enumerates 5-state Turing machines in Tree Normal Form, Section 3 and Table 2. Each enumerated machine goes through the $S(5)$ pipeline, Table 3, where the halting problem from all-0 tape of each machine is solved using a decider (or verifier), Section 4, unless the machine is one of the 13 Sporadic Machines, Section 5, for which individual Coq proofs of nonhalting are provided. When encountering a halting machine, the proof checks that it halts before 47,176,870 steps, giving $S(5) \leq 47{,}176{,}870$, see `Lemma BB5_upperbound`, and, using the 5-state champion (Figure 2), see `Lemma BB5_lowerbound`, the proof concludes $S(5) = 47{,}176{,}870$. Thanks to this proof, the 5-state champion becomes the winner of the 5th Busy Beaver competition, Figure 2. □

Extracting machines. The essential output of The Busy Beaver Challenge is the list of all 181,385,789 TNF-enumerated 5-state machines together with their halting status and method used to determine it. Find the list at https://docs.bbchallenge.org/CoqBB5_release_v1.0.0/. This list was computed by *extracting* the Coq proof to OCaml, which means that all the Coq-implemented algorithms where automatically transcribed in a trusted way to OCaml by the Coq engine. From there, print statements were added to the OCaml code which, once ran, produced the list. If given an arbitrary 5-state Turing machine, computing TNF normalisation (see Section 3) and then performing lookup in the list allows to determine the halting status of the machine from all-0 tape.

Using this Coq-verified list, any "observable" metric on 5-state Turing machines such as Radó's Σ (see Section 1) can be computed:

Theorem 6.1. $\Sigma(5) = 4{,}098$.

Proof. The winning machine for $\Sigma(5)$ is the same as for $S(5)$, Figure 2. This metric was computed from the Coq-extracted list of all TNF-enumerated 5-state Turing machines, see above. Three agreeing independent reproductions of the computation were asked to ensure there was no mistake made. □

Another similar observable, called space(n) [3] (also called $\mathrm{BB}_{\mathrm{SPACE}}(n)$ [58]) is the maximum number of tape cells that an n-state Turing machine may scan before it halts. This observable is similar to S in the sense that, if a machine visits space$(n)+1$ tape cells, we know it will never halt (from the all-0 tape). We get:

Theorem 6.2. space$(5) = 12{,}289$.

Proof. The winning machine for space(5) is the same as for $S(5)$, Figure 2. Same method as for Σ, Theorem 6.1. □

Coq-BB5 also computes S for 2-state 4-symbol Turing machines:

Theorem 1.2 (Coq-BB5: `Lemma BB2x4_value`)**.** $S(2,4) = 3{,}932{,}974$.

Proof. Similarly to Theorem 1.1, the Coq proof enumerates 2-state 4-symbol machines in (almost) Tree Normal Form, see Section 3 and Table 2. Then the $S(2,4)$ pipeline, Table 4, which consists only of deciders (Section 4) is applied to solve the halting problem from all-0 tape of all the enumerated machines. The proof keeps track of the maximum number of steps reached by halting machines and eventually concludes $S(2,4) = 2{,}154{,}217$. Thanks to this proof, the 2-state 4-symbol champion (Figure 3) becomes the winner among all 2-state 4-symbol machines. □

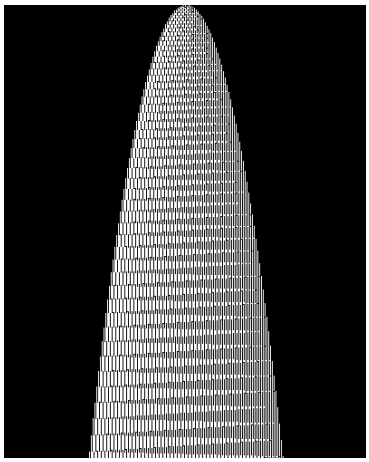
Additionally, as illustrated in Table 1, Coq-BB5 also provide Coq proofs for previously known values of S, including $S(4)$ of which original proof [8] had slight uncertainties – see Section 1:

Theorem 6.3 (Confirmation of Brady's result [8])**.** $S(4) = 107$.

Proof. Same as Theorem 1.1, using the $S(4)$ pipeline, Table 5, which consists only of deciders (Section 4), i.e. no individual proofs of nonhalting. □

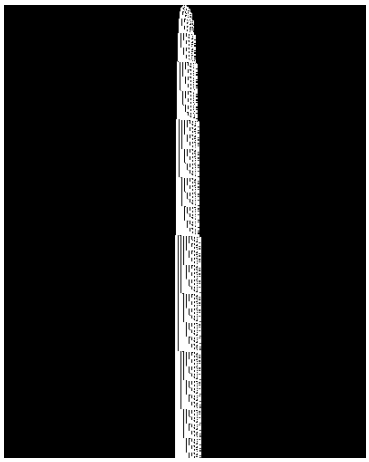
[44] See https://rocq-prover.org/doc/v8.9/stdlib/Coq.Logic.FunctionalExtensionality.html.

7 Zoology



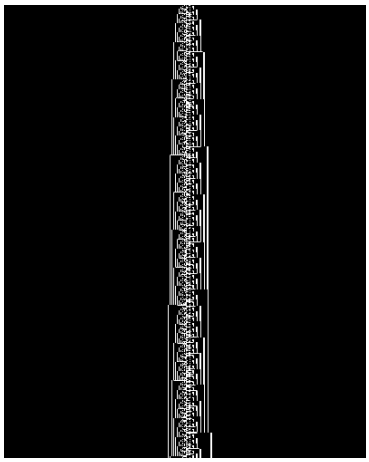
Bouncers

1RB0LE_0RC1RE_1LD1RA_0LA---_0RC0LB



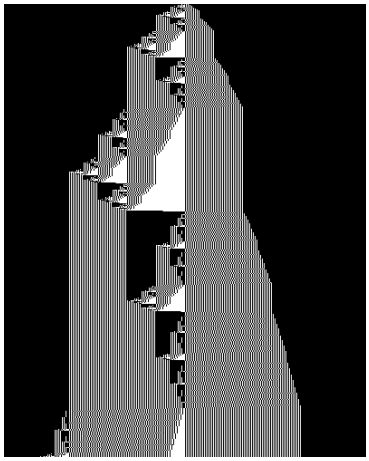
Counters

1RB1LA_0LA0RC_0LC1RD_1RE0LA_0RB---



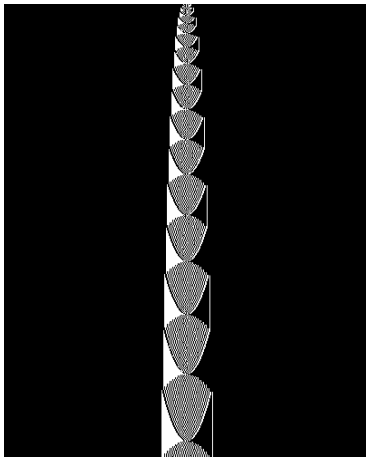
Double Counters

1RB1LE_1LC0RB_1RD0LC_1LA1RB_1LB---



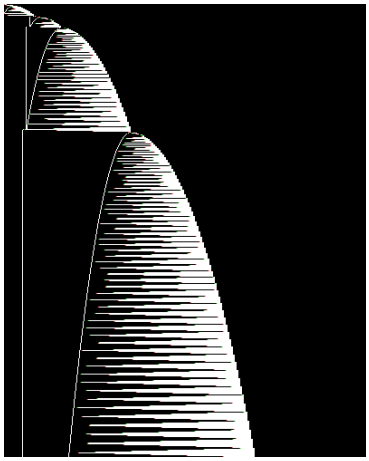
Fractals

1RB0RB_1LC0RA_1LD0LB_0LE---_1RE1LB



Helices

1RB0RE_0RC0RA_1LD---_1LA0LB_1RA0LC



Bells

Current 6-state champion [31]

Figure 17: Main zoological families that were identified among 5-state Turing machines, together with Cyclers and Translated Cyclers which are not illustrated here (see Section 4.2).

As we were *computing in the wild*, we identified several *zoological families* among 5-state Turing machines:

1. **Cyclers and Translated Cyclers**; Section 4.2. Translated Cyclers is the most common *species* among 5-state machines – estimated to be about 80% of TNF-enumerated machines.

2. **Bouncers**; Figure 17. This family consists of machines that populate the tape with linearly-expanding patterns while bouncing back and forth from tape extremities. Bouncers have been formally defined and efficient deciders (not used in Coq-BB5) have been crafted to detect them [62].

3. **Counters**[45]; Figure 17. Counters are machines that enumerate machines in a basis bigger than 1. There is a rich variety of 5-state counters, counting in all sorts of bases: base 2 (Figure 13), base 3, base 3/2, base Fibonacci (Section 5) and Fibonacci variants $A(n) = A(n-1) + A(n-3)$ etc.[46]

4. **Double Counters**; Figure 17. Double Counters implement two independent counters, Skelet #10 is an example (Section 5). The example of Figure 17 is a base-2 counter on the left-side of the tape and base-3 counter on the right-side.

[45] Also referred to as *exponential counters* because, since they count in a basis > 1, it takes them exponentially long to expand the tape of one digit.

[46] Classifying 5-state counters would be a beautiful project.

5. **Fractals**; Figure 17. Fractal machines are loosely define as machines whose space-time diagrams draw a fractal pattern. There some hybrids, such as this Sierpiński triangle growing off the side of a bouncer.

6. **Helices**; Figure 17. Helices are loosely defined as Turing machines whose space-time diagrams resembles a double helix. Helices required big nonregular certificates of nonhalting, see Section 4.6.

7. **Bells**; Figure 17. Helices are loosely defined as Turing machines whose space-time diagrams resembles a succession of bells. The 5-state winner (Figure 2), the 2-state 4-symbol winner (Figure 3), and, the current 6-state champion (Figure 17) fit this category.

Although useful for talking about Turing machines, these families, especially when there are loosely defined, are not to be taken too seriously: (i) widely different behaviors can be implemented while maintaining a similar space-time diagram *silhouette* and (ii) the zoological effort can quickly become vain given all the possible hybrids (e.g. bouncer-counter) and variations within the same family, especially as the number of states increases. Attesting to the limits of a zoological effort, here are excentric 5-state machines: translated counter, "pachinko", and, "toboggan".

References

[1] S. Aaronson. The Busy Beaver Frontier. *SIGACT News*, 51(3):32–54, Sept. 2020. https://www.scottaaronson.com/papers/bb.pdf.

[2] bbchallenge wiki. Champions. https://wiki.bbchallenge.org/wiki/Champions, 2025. [Online; accessed 11-May-2025].

[3] A. M. Ben-Amram, B. A. Julstrom, and U. Zwick. A note on busy beavers and other creatures. *Mathematical systems theory*, 29(4):375–386, Aug 1996.

[4] J. Blanchard. Finite Automata Reduction (FAR). https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-finite-automata-reduction, 2022.

[5] M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs*, pages 362–377, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] M. Bolan, J. Breitner, J. Brox, M. Carneiro, M. Dvořák, A. Goens, A. Hill, H. Husum, Z. Kocsis, B. L. Floch, L. Luccioli, A. Meiburg, P. Monticone, G. Paolini, B. Reinke, D. Renshaw, M. Rossel, C. Roux, J. Scanvic, S. Srinivas, A. R. Tadipatri, T. Tao, V. Tsyrklevich, D. Weber, and F. Zheng. The Equational Theories Project: Advancing Collaborative Mathematical Research at Scale, 2025. In preparation.

[7] A. H. Brady. *Solutions of restricted cases of the halting problem applied to the determination of particular values of a non-computable function.* PhD thesis, Oregon State University, 1964.

[8] A. H. Brady. The determination of the value of rado's noncomputable function $\Sigma(k)$ for four-state turing machines. *Mathematics of Computation*, 40(162):647–665, 1983.

[9] A. H. Brady and t. M. o. Life. The busy beaver game and the meaning of life. In *The Universal Turing Machine: A Half-Century Survey*. Oxford University Press, 03 1990.

[10] D. Briggs. Turing. 2010. https://github.com/danbriggs/Turing.

[11] Code Golf Addict. list27.txt. https://gist.github.com/anonymous/a64213f391339236c2fe31f8749a0df6, 2016.

[12] T. Coquand, J. Gallier, and L. Cedex. A Proof of Strong Normalization For the Theory of Constructions Using a Kripke-Like Interpretation. 04 1999.

[13] R. Cuninghame-Green. Minimax algebra and applications. *Fuzzy Sets and Systems*, 41(3):251–267, 1991.

[14] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[15] A. DUBICKAS. On integer sequences generated by linear maps. *Glasgow Mathematical Journal*, 51(2):243–252, 2009.

[16] ETP. Equational theories project. https://github.com/teorth/equational_theories, 2024.

[17] F. Faase. Symbolic TM. https://github.com/FransFaase/SymbolicTM, 2022.

[18] N. Fenner. bbchallenge-regexy-decider. hhttps://github.com/Nathan-Fenner/bbchallenge-regexy-decider, 2022.

[19] G. Georgiev. Busy Beaver nonregular machines for class TM(5). https://skelet.ludost.net/bb/nreg.html, 2003. Accessed: 2025-04-03.

[20] G. Georgiev. Busy Beaver prover - `bbfind`. https://skelet.ludost.net/bb/, 2003. Accessed: 2024-11-25.

[21] T. Gowers and M. Nielsen. Massively collaborative mathematics. *Nature*, 461(7266):879–881, Oct 2009.

[22] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. *SIGPLAN Not.*, 37(9):235–246, Sept. 2002.

[23] T. HALES, M. ADAMS, G. BAUER, T. D. DANG, J. HARRISON, L. T. HOANG, C. KALISZYK, V. MAGRON, S. MCLAUGHLIN, T. NGUYEN, and et al. A formal proof of the kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017.

[24] J. Harland. The busy beaver, the placid platypus and other crazy creatures. In *Proceedings of the 12th Computing: The Australasian Theroy Symposium - Volume 51*, CATS '06, page 79–86, AUS, 2006. Australian Computer Society, Inc.

[25] J. Hertel. Computing the Uncomputable Rado Sigma Function. *The Mathematica Journal*, 11, 11 2009.

[26] hipparcos. turing_machine. https://github.com/jhuang97/turing_machine, 2025.

[27] Iijil. Bruteforce-CTL. https://github.com/Iijil1/Bruteforce-CTL, 2022.

[28] Iijil. Finned 3 is irregular. https://discuss.bbchallenge.org/t/10756090-is-irregular/137, 2023.

[29] Iijil1. Iijil1/mitmwfar: v1.0.0, Feb. 2025.

[30] Johannes Riebel. The Undecidability of BB(748). Bachelor's thesis, 2023. https://www.ingo-blechschmidt.eu/assets/bachelor-thesis-undecidability-bb748.pdf.

[31] P. Kropitz. BB(6, 2) > 4^4^4^4^7, 2022. https://groups.google.com/g/busy-beaver-discuss/c/-zjeW6y8ER4/m/ZBuLvbVOAgAJ.

[32] G. Laffite and P. C. The fabric of small Turing machines. In *Computation and Logic in the Real World, Proceedings of the Third Conference on Computabiliy in Europe, CiE '07*, pages 219–227, Berlin, Heidelberg, 2007. Springer-Verlag.

[33] J. C. Lagarias. The 3x + 1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985.

[34] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, page 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] Y. Leng. lengyijun/goldbach_tm: 25-state turing machine, Jan. 2025.

[36] S. Ligocki. CTL Filter. Blog: https://www.sligocki.com/2022/06/10/ctl.html, 2022. Accessed: 2023-03-20.

[37] S. Ligocki. Shift Overflow Counters. Blog: hhttps://www.sligocki.com/2023/02/05/shift-overflow.html, 2023. Accessed: 2025-04-04.

[38] S. Ligocki. Skelet #1 is infinite ... we think. Blog: https://www.sligocki.com/2023/03/13/skelet-1-infinite.html, 2023. Accessed: 2024-02-11.

[39] S. Ligocki. Skelet #1: What I Know. Blog: https://www.sligocki.com/2023/02/25/skelet-1-wip.html, 2023. Accessed: 2025-04-07.

[40] S. Ligocki. Skelet #10: Double Fibonacci Counter. Blog: https://www.sligocki.com/2023/03/14/skelet-10.html, 2023. Accessed: 2025-04-07.

[41] S. Lin. *Computer studies of Turing machine problems*. PhD thesis, Ohio State University, Graduate School, 1963.

[42] H. Marxen and J. Buntrock. Attacking the Busy Beaver 5. *Bull. EATCS*, 40:247–251, 1990.

[43] mei. busycoq. https://github.com/meithecatte/busycoq/blob/master/verify/Skelet1.v, 2023.

[44] P. Michel. The Busy Beaver Competitions. https://bbchallenge.org/∼pascal.michel/bbc.html [Accessed 04-08-2024].

[45] P. Michel. Small turing machines and generalized busy beaver competition. *Theoretical Computer Science*, 326(1):45–56, 2004.

[46] P. Michel. The Busy Beaver Competition: a historical survey. Technical report, Sept. 2019. https://arxiv.org/abs/0906.3749v6.

[47] Nathan Fenner. bbchallenge Dafny deciders. https://github.com/Nathan-Fenner/bbchallenge-dafny-deciders, 2022.

[48] Nathan Fenner. n-GRAM CPS Decider. https://github.com/Nathan-Fenner/bb-simple-n-gram-cps, 2023.

[49] Owen Kellett. A multi-faceted attack on the busy beaver problem. Master's thesis, 2005. https://homepages.hass.rpi.edu/heuveb/research/BB/downloads/OwenThesis.pdf.

[50] S. O'Rear. metamath - ZF. https://github.com/sorear/metamath-turing-machines/blob/master/zf2.nql, 2017.

[51] Pavel Kropitz. bbc. https://github.com/univerz/bbc/tree/no1, 2023.

[52] T. Radó. On non-computable functions. *Bell System Technical Journal*, 41(3):877–884, 1962. https://archive.org/details/bstj41-3-877/mode/2up.

[53] T. Radó. On a simple source for non-computable functions. In *Proceedings of the Symposium on Mathematical Theory of Automata (New York)*. Polytechnic Press of the polytechnique Institue of Brooklyn, 1963.

[54] K. Ross, O. Kellett, B. van Heuveln, and S. Bringsjord. A new-millenium attack on the busy beaver problem. 2005. https://homepages.hass.rpi.edu/heuveb/research/BB/downloads/superpaper.pdf.

[55] savask. Analysis of Skelet's machine 17. https://chrisxudoesmath.com/papers/skelet17_savasks_analysis.pdf, 2023. Accessed: 2025-04-07.

[56] savask. RepWL Haskell implementation. https://github.com/savask/turing/blob/main/Repeat.hs, 2024.

[57] T. Stérin and D. Woods. Hardness of Busy Beaver Value BB(15). In L. Kovács and A. Sokolova, editors, *Reachability Problems*, pages 120–137, Cham, 2024. Springer Nature Switzerland.

[58] T. Stérin. bbchallenge.org, initial release, Mar. 2022.

[59] T. Tao. A pilot project in universal algebra to explore new ways to collaborate and use machine assistance? Blog: https://terrytao.wordpress.com/2024/09/25/a-pilot-project-in-universal-algebra-to-explore-new-ways-to-collaborate-and-use-machine-assista 2024. Accessed: 2025-05-11.

[60] T. C. D. Team. The coq proof assistant, Dec. 2024.

[61] L. Teodorescu, G. Baudart, E. J. G. Arias, and M. Lelarge. NLIR: Natural language intermediate representation for mechanized theorem proving. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024.

[62] The bbchallenge Collaboration, J. Blanchard, K. Deka, N. Fenner, T. Guilfoyle, Iijil, M. Kądziołka, P. Kropitz, S. Ligocki, P. Michel, M. Naściszewski, and T. Stérin. Turing machines deciders, part I, Apr. 2025. https://arxiv.org/abs/2504.20563.

[63] Tony Guilfoyle. FAR C++ reproduction. https://github.com/TonyGuil/bbchallenge/tree/main/FAR, 2023.

[64] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, Jan 2024.

[65] Tristan Stérin. FAR Python reproduction. https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-finite-automata-reduction-reproduction, 2023.

[66] Tristan Stérin. RepWL Python implementation. https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-repeated-word-list-reproduction, 2024.

[67] J. Tromp. Busy Beaver for lambda calculus BBλ. https://oeis.org/A333479, 2020. Entry A333479 in The On-Line Encyclopedia of Integer Sequences.

[68] Wikipedia. Cycle detection — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cycle%20detection&oldid=1265904359, 2025. [Online; accessed 30-January-2025].

[69] Wikipedia. Fast-growing hierarchy — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Fast-growing_hierarchy, 2025. [Online; accessed 11-May-2025].

[70] Wikipedia. Zeckendorf's theorem — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Zeckendorf's%20theorem&oldid=1242695626, 2025. [Online; accessed 07-April-2025].

[71] Z. Wu, S. Huang, Z. Zhou, H. Ying, J. Wang, D. Lin, and K. Chen. Internlm2.5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems, 2024.

[72] C. Xu. Skelet #17 and the fifth Busy Beaver number, 2024. https://arxiv.org/abs/2407.02426.

[73] A. Yedidia and S. Aaronson. A relatively small Turing machine whose behavior is independent of set theory. *Complex Systems*, 25(4):297–328, Dec. 2016.

[74] D. Yuan and J. Blanchard. Skelet 17 is irregular. https://cosearch.bbchallenge.org/contribution/rgr5sxom, 2024.

[75] S. A. Yuri Matiyasevich, Stefan O'Rear. metamath - Riemann Hypothesis. https://github.com/sorear/metamath-turing-machines/blob/master/riemann-matiyasevich-aaronson.nql, 2016.

A Author Contributions

The bbchallenge Collaboration: $S(5)$ credits. The following contributions resulted in the determination of the fifth Busy Beaver value and in the better understanding of the landscape of small Busy Beaver values (Table 1): mxdys (Coq-BB5, Loops, RepWL); Nathan Fenner, Georgi Georgiev a.k.a Skelet, savask, mxdys (NGramCPS); Justin Blanchard, Mateusz Naściszewski, Konrad Deka (FAR); Iijil (WFAR); mei (`busycoq`); Shawn Ligocki, Jason Yuen, mei (Sporadic Machines "Shift Overflow Counters"); Shawn Ligocki, Pavel Kropitz, mei (Sporadic Machine "Skelet #1"); savask, Chris Xu, mxdys (Sporadic Machine "Skelet #17"); Shawn Ligocki, Dan Briggs, mei (Sporadic Machine "Skelet #10"); Justin Blanchard, mei (Sporadic Machines "Finned Machines"); Shawn Ligocki, Daniel Yuan, mxdys, Matthew L. House, Rachel Hunter, Jason Yuen ("Cryptids"); Yannick Forster, Théo Zimmermann (Coq review); Yannick Forster (Coq optimisation); Tristan Stérin (bbchallenge.org); Tristan Stérin, Justin Blanchard (paper writing).

- The bbchallenge Collaboration, bbchallenge.org, `bbchallenge@bbchallenge.org`
- Justin Blanchard, `UncombedCoconut@gmail.com`
- Dan Briggs, Affiliation?, email?
- Konrad Deka, `deka.konrad@gmail.com`
- Nathan Fenner, `nfenneremail@gmail.com`
- Yannick Forster, INRIA Paris, `yannick.forster@inria.fr`
- Georgi Georgiev (Skelet), Sofia University, Faculty of Mathematics and Informatics, `skeleta@gmail.com`
- Rachel Hunter, `racheline@bbchallenge.org`
- Matthew L. House, `mattlloydhouse@gmail.com`
- Iijil, `hheussen@web.de`
- Maja Kądziołka, `bb@compilercrim.es`
- Pavel Kropitz, `uni@bbchallenge.org`
- Shawn Ligocki, `sligocki@gmail.com`
- mxdys, `mxdys@bbchallenge.org`
- Mateusz Naściszewski
- savask
- Tristan Stérin, PRGM DEV, `tristan@prgm.dev`
- Chris Xu, UC San Diego, `chx007@ucsd.edu`
- Jason Yuen, `jason_yuen2007@hotmail.com`
- Théo Zimmermann, LTCI, Télécom Paris, Institut Polytechnique de Paris, France, `theo.zimmermann@telecom-paris.fr`.

Acknowledgement. The bbchallenge Collaboration is not limited to the above contributors but regroups all those who participated in the bbchallenge's discussions through all our channels (Discord chat, forum, wiki, GitHub, emails) and in particular we thank: Frans Faase, Tony Guilfoyle, Nick Howell, Alexandre Jouandin, Carl Kadie, Dawid Loranc, Terry J. Ligocki, Heiner Marxen, Pascal Michel, Milo Mighdoll (milomg), Sébastien Ohleyer, star, tomtom2357, Valentin, Daniel Yuan.

We'd also like to thank the following people who helped disseminate the bbchallenge project: Damien Woods, Dave Doty, Eric E. Severson, Scott Aaronson, Ben Brubaker, and Léo Ramaën.

B Busy Beaver champions and winners

TODO: essentially take from https://wiki.bbchallenge.org/wiki/Champions and from Pascal Michel's website https://bbchallenge.org/~pascal.michel/index.html.

C Exact pipelines

TODO: essentially take from Coq-BB5 READMEs "Results" sections, e.g. https://github.com/ccz181078/Coq-BB5/tree/main/CoqBB5/BB5#results.

D Cryptids

TODO: essentially take form https://wiki.bbchallenge.org/wiki/Cryptids.

E Use of AI

The $S(5)$ proof (2022 - 2024) is roughly concomitant with the striking progresses of Large Language Models in AI. We disclose their use – or lack thereof – in our project:

- **Research co-pilot.** The idea of using Bellman-Ford algorithm in WFAR (Section 4.6) was given by a LLM (ChatGPT 4o). **Note:** this was found at the time of writing the paper, roughly 2 years after the initial release of the WFAR algorithm and 1 year after its integration in Coq-BB5, both of which did not use this trick and limited themselves to a less general class of weighted automata [29].
- **Code co-pilot.** AI-based code completion **was not** used in Coq-BB5. AI-based code completion is unlikely to have been used for deciders written before Coq-BB5 (see https://wiki.bbchallenge.org/wiki/Code_repositories) as most of them were developed before AI-based code completion was mature. AI-based code completion was used to study Coq-BB5 in preparation of this paper (for instance: accelerating the reproduction of some algorithms, or translating Coq code to Python in order to understand it better).
- **Writing co-pilot.** AI based copy editing was used (i) to verify the spelling and grammar of the text and, (ii) to compress two lengthy paragraphs of human-written text (marked with a `%SIA` comment in the LaTeX source, original paragraphs are kept commented). AI was **extensively used** to make and improve figures, mainly through LaTeX and `tikz` code generation, often prompting the AI with a hand-drawn layout of what was wanted and then iterating together – e.g. Figure 4, Figure 14, Figure 16, Figure 17.