bbchallenge paper

bbchallenge's contributors

Abstract

TODO

Contents

1 Introduction

1.1 Busy beaver functions

1.2 The busy beaver scale: weighing open problems in mathematics

1.3 Formal verification: Coq

2 The road to S(4) and S(2,4)

2.1 Cyclers

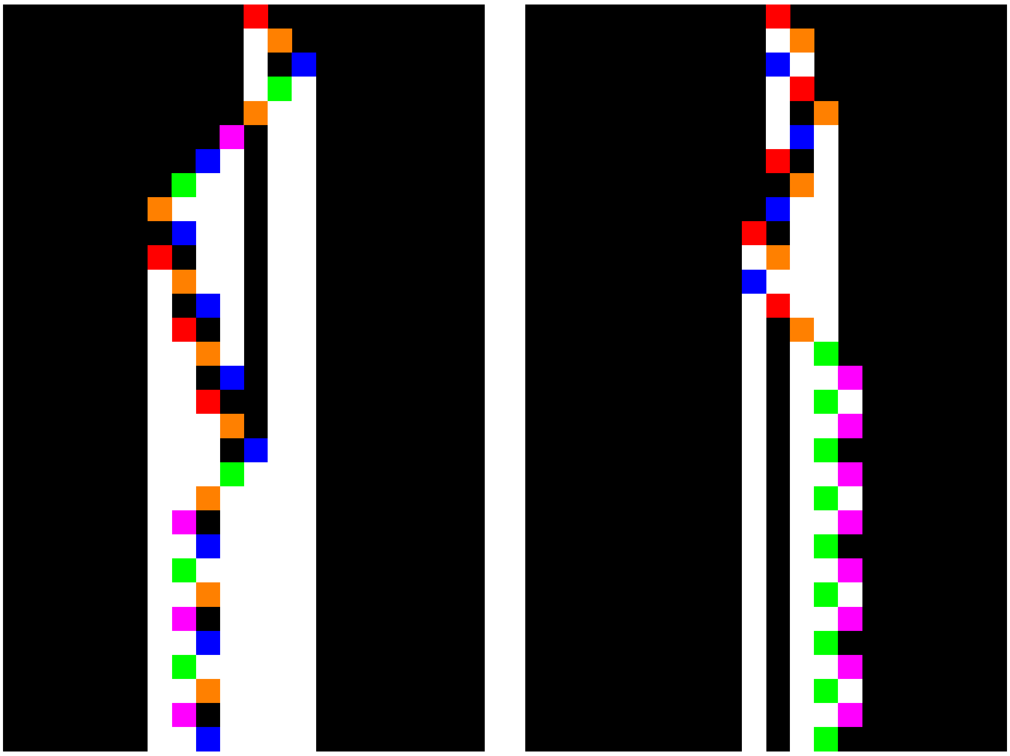


Figure 1: Space-time diagrams of the 30 first steps of bbchallenge's machines #279,081 (left) and #4,239,083 (right) which are both "Cyclers": they eventually repeat the same configuration for ever. Access the machines at `https://bbchallenge.org/279081` and `https://bbchallenge.org/4239083`.

The goal of this decider is to recognise Turing machines that cycle through the same configurations forever. Such machines never halt. The method is simple: remember every configuration seen by a machine and return `true` if one is visited twice. A time limit (maximum number of steps) is also given for running the test in practice: the algorithm recognises any machine whose cycle fits within this limit[1].

Example 2.1. Figure 1 gives the space-time diagrams of the 30 first iterations of two "Cyclers" machines: bbchallenge's machines #279,081 (left) and #4,239,083 (right). Refer to `https://bbchallenge.org/279081` and `https://bbchallenge.org/4239083` for their transition tables. From these space-time diagrams we see that the machines eventually repeat the same configuration.

2.1.1 Pseudocode

We assume that we are given a Turing Machine type **TM** that encodes the transition table of a machine as well as a procedure **TuringMachineStep**(machine,configuration) which computes the next configuration of a Turing machine from the given configuration or **nil** if the machine halts at that step. The pseudocode is given in Algorithm 1.

2.1.2 Correctness

Theorem 2.2. Let $\mathcal{M}$ be a Turing machine and $t \in \mathbb{N}$ a time limit. Let c_0 be the initial configuration of the machine. There exists $i \in \mathbb{N}$ and $j \in \mathbb{N}$ such that $c_0 \vdash^i c_i \vdash^{j-i} c_i$ with $i < j \leq t$ if and only if DECIDER-CYCLERS($\mathcal{M}$,t) returns `true` (Algorithm 1).

Proof. This follows directly from the behavior of DECIDER-CYCLERS($\mathcal{M}$,t): all configurations from c_0 to c_t are recorded and the algorithm returns `true` if and only if one is visited twice. This mathematically translates to there exists $i \in \mathbb{N}$ and $j \in \mathbb{N}$ such that $c_0 \vdash^i c_i \vdash^{j-i} c_i$ with $i < j \leq t$, which is what we want. Index i corresponds to the first time that c_i is seen (l.14 in Algorithm 1) while index j corresponds to the second time that c_i is seen (l.12 in Algorithm 1). □

[1] In practice, for machines with 5 states the decider was run with 1000 steps time limit.

Algorithm 1 DECIDER-CYCLERS

```
struct Configuration {
    State state
    int headPosition
    int → int tape
}

procedure bool DECIDER-CYCLERS(TM machine,int timeLimit)
    Configuration currConfiguration = {.state = A, .headPosition = 0, .tape = {0:0}}
    Set<Configuration> configurationsSeen = {}
    int currTime = 0
    while currTime ≤ timeLimit do
        if currConfiguration in configurationsSeen then
            return true
        configurationsSeen.insert(currConfiguration)
        currConfiguration = TuringMachineStep(machine,currConfiguration)
        currTime += 1
        if currConfiguration == nil then
            return false //machine has halted, it is not a Cycler
    return false
```

Corollary 2.3. Let $\mathcal{M}$ be a Turing machine and $t \in \mathbb{N}$ a time limit. If DECIDER-CYCLERS($\mathcal{M}$,t) returns `true` then the behavior of $\mathcal{M}$ from all-0 tape has been decided: $\mathcal{M}$ does not halt.

Proof. By Theorem 2.2, there exists $i \in \mathbb{N}$ and $j \in \mathbb{N}$ such that $c_0 \vdash^i c_i \vdash^{j-i} c_i$ with $i < j \leq t$. It follows that for all $k \in \mathbb{N}$, $c_0 \vdash^{i+k(j-i)} c_i$. The machine never halts as it will visit c_i infinitely often. □

2.1.3 Results

The decider was coded in `golang` and is accessible at this link: `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-cyclers`.

The decider found 11,229,238 "Cyclers", out of 88,664,064 machines in the seed database of the Busy Beaver Challenge (c.f. `https://bbchallenge.org/method#seed-database`). Time limit was set to 1000 and an additional memory limit (max number of visited cells) was set to 500. More information about these results are available at: `https://discuss.bbchallenge.org/t/decider-cyclers/33`.

2.2 Translated cyclers

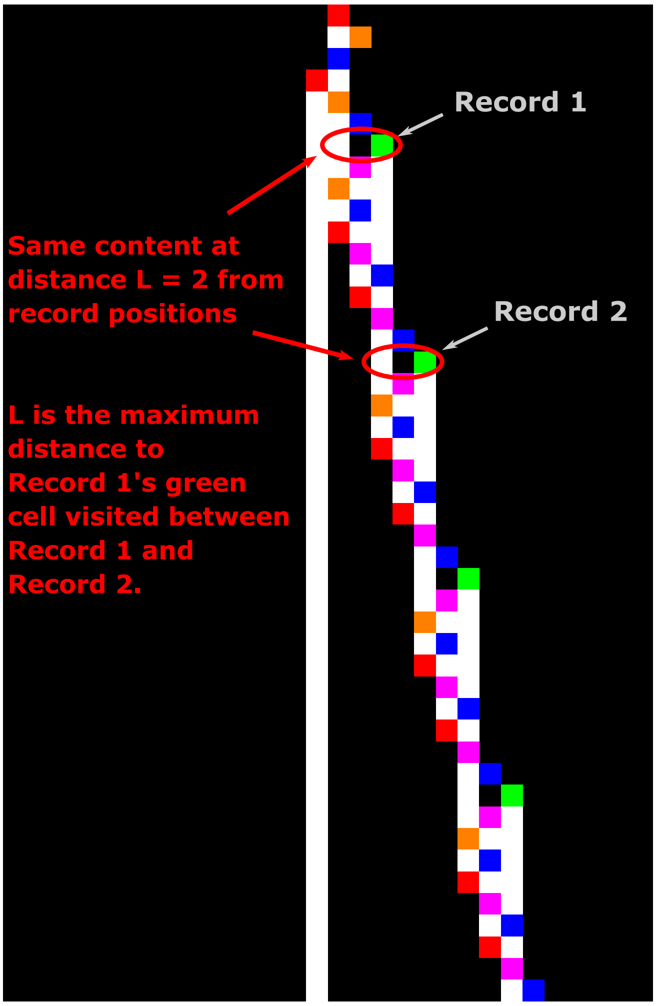


Figure 2: Example "Translated cycler": 45-step space-time diagram of bbchallenge's machine #44,394,115. See `https://bbchallenge.org/44394115`. The same bounded pattern is being translated to the right forever. The text annotations illustrate the main idea for recognising "Translated Cyclers": find two configurations that break a record (i.e. visit a memory cell that was never visited before) in the same state (here state D) such that the content of the memory tape at distance L from the record positions is the same in both record configurations. Distance L is defined as being the maximum distance to record position 1 that was visited between the configuration of record 1 and record 2.

The goal of this decider is to recognise Turing machines that translate a bounded pattern forever. We call such machines "Translated cyclers". They are close to "Cyclers" (Section 2.1) in the sense that they are only repeating a pattern but there is added complexity as they are able to translate the pattern in space at the same time, hence the decider for Cyclers cannot directly apply here.

The main idea for this decider is illustrated in Figure 2 which gives the space-time diagram of a "Translated cycler": bbchallenge's machine #44,394,115 (c.f. `https://bbchallenge.org/44394115`). The idea is to find two configurations that break a record (i.e. visit a memory cell that was never visited before) in the same state (here state D) such that the content of the memory tape at distance L from the record positions is the same in both record configurations. Distance L is defined as being the maximum distance to record position 1 that was visited between the configuration of record 1 and record 2. In those conditions, we can prove that the machine will never halt.

The translated cycler of Figure 2 features a relatively simple repeating pattern and transient pattern (pattern occurring before the repeating patterns starts). The translated cycler of Figure 3 features a significantly more complex pattern. The method for detecting the behavior is the same but more resources are needed.

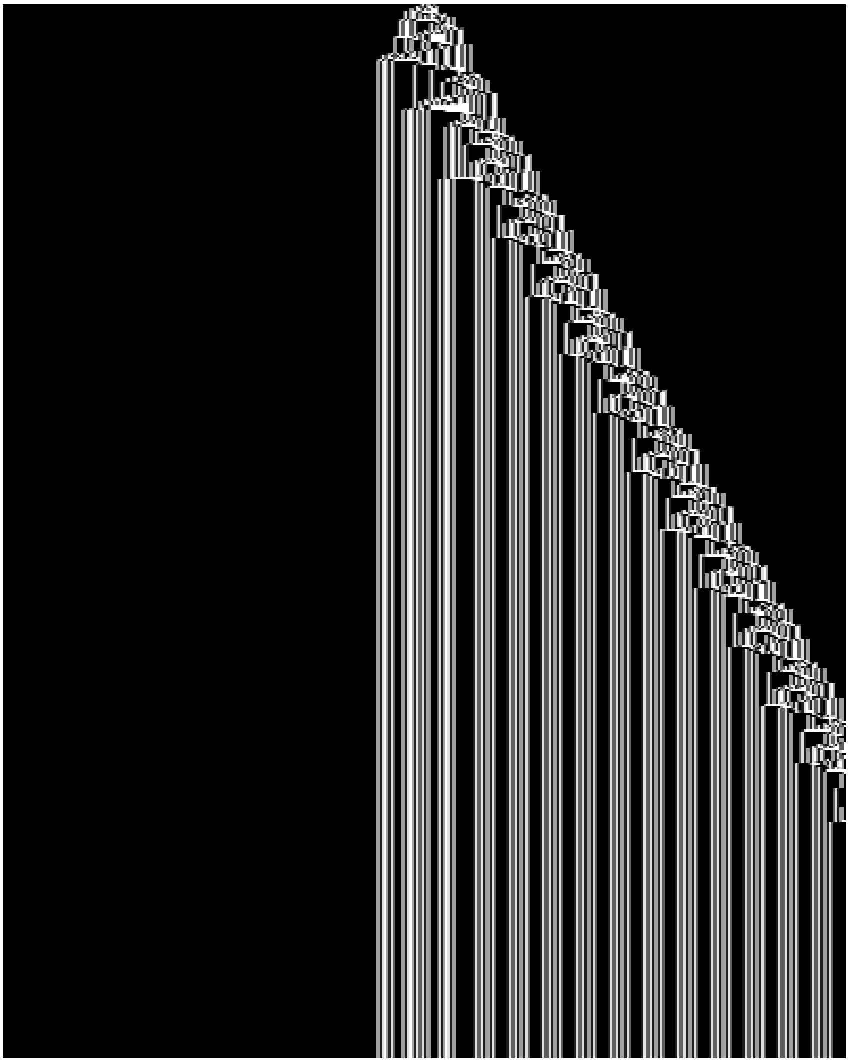


Figure 3: More complex “Translated cycler”: 10,000-step space-time diagram (no state colours) of bbchallenge’s machine #59,090,563. See `https://bbchallenge.org/59090563`.

2.2.1 Pseudocode

We assume that we are given a Turing Machine type **TM** that encodes the transition table of a machine as well as a procedure **TuringMachineStep**(machine,configuration) which computes the next configuration of a Turing machine from the given configuration or **nil** if the machine halts at that step.

One minor complication of the technique described above is that one has to track record-breaking configurations on both sides of the tape: a configuration can break a record on the right or on the left. Also, in order to compute distance L (see above or Definition 2.5) it is useful to add to memory cells the information of the last time step at which it was visited.

We also assume that we are given a routine GET-EXTREME-POSITION(tape,sideOfTape) which gives us the rightmost or leftmost position of the given tape (well defined as we always manipulate finite tapes).

Algorithm 2 DECIDER-TRANSLATED-CYCLERS

```
const int RIGHT, LEFT = 0, 1
struct ValueAndLastTimeVisited {
    int value
    int lastTimeVisited
}
struct Configuration {
    State state
    int headPosition
    int → ValueAndLastTimeVisited tape
}

procedure bool DECIDER-TRANSLATED-CYCLERS(TM machine,int timeLimit)
    Configuration currConfiguration = {.state = A, .headPosition = 0, .tape = {0:{.value = 0, .lastTimeVisited = 0}}}
    // 0: right records, 1: left records
    List<Configuration> recordBreakingConfigurations[2] = [[],[]]
    int extremePositions[2] = [0,0]
    int currTime = 0
    while currTime < timeLimit do
        int headPosition = currConfiguration.headPosition
        currConfiguration.tape[headPosition].lastTimeVisited = currTime
        if headPosition > extremePositions[RIGHT] or headPosition < extremePositions[LEFT] then
            int recordSide = (headPosition > extremePositions[RIGHT]) ? RIGHT : LEFT
            extremePositions[recordSide] = headPosition
            if AUX-CHECK-RECORDS(currConfiguration, recordBreakingConfigurations[recordSide], recordSide) then
                return true
            recordBreakingConfigurations[recordSide].append(currConfiguration)
        currConfiguration = TuringMachineStep(machine,currConfiguration)
        currTime += 1
        if currConfiguration == nil then
            return false //machine has halted, it is not a Translated Cycler
    return false
```

2.2.2 Correctness

Definition 2.4 (record-breaking configurations)**.** Let $\mathcal{M}$ be a Turing machine and c_0 its busy beaver initial configuration (i.e. state is 0, head position is 0 and tape is all-0). Let c be a configuration reachable from c_0, i.e. $c_0 \vdash^* c$. Then c is said to be *record-breaking* if the current head position had never been visited before. Records can be broken to the *right* (positive head position) or to the left (negative head position).

Definition 2.5 (Distance L between record-breaking configurations)**.** Let $\mathcal{M}$ be a Turing machine and r_1, r_2 be two record-breaking configurations on the same side of the tape at respective times t_1 and t_2 with $t_1 < t_2$. Let p_1 and p_2 be the tape positions of these records. Then, distance L between r_1 and r_2 is

Algorithm 3 COMPUTE-DISTANCE-L and AUX-CHECK-RECORDS

```
procedure int COMPUTE-DISTANCE-L(Configuration currRecord, Configuration olderRecord,
int recordSide)
    int olderRecordPos = olderRecord.headPosition
    int olderRecordTime = olderRecord.tape[olderRecordPos].lastTimeVisited
    int currRecordTime = currRecord.tape[currRecord.headPosition].lastTimeVisited
    int distanceL = 0
    for int pos in currRecord.tape do
        if pos > olderRecordPos and recordSide == RIGHT then continue
        if pos < olderRecordPos and recordSide == LEFT then continue
        int lastTimeVisited = currRecord.tape[pos].lastTimeVisited
        if lastTimeVisited ≥ olderRecordTime and lastTimeVisited ≤ currRecordTime then
            distanceL = max(distanceL,abs(pos-olderRecordPos))
    return distanceL

procedure bool AUX-CHECK-RECORDS(Configuration currRecord, List<Configuration> older-
Records, int recordSide)
    for Configuration olderRecord in olderRecords do
        if currRecord.state != olderRecord.state then
            continue
        int distanceL = COMPUTE-DISTANCE-L(currRecord,olderRecord,recordSide)
        int currExtremePos = GET-EXTREME-POSITION(currRecord.tape,recordSide)
        int olderExtremePos = GET-EXTREME-POSITION(olderRecord.tape,recordSide)
        int step = (recordSide == RIGHT) ? -1 : 1
        bool isSameLocalTape = true
        for int offset = 0; abs(offset) ≤ distanceL; offset += step do
            if currRecord.tape[currExtremePos+offset].value !=
olderRecord.tape[olderExtremePos+offset].value then
                isSameLocalTape = false
                break
        if isSameLocalTape then
            return true
    return false
```

defined as $\max\{|p_1 - p|\}$ with p any position visited by $\mathcal{M}$ between t_1 and t_2 that is not beating record p_1 (i.e. $p \leq p_1$ for a record on the right and $p \geq p_1$ for a record on the left).

Lemma 2.6. Let $\mathcal{M}$ be a Turing machine. Let r_1 and r_2 be two configurations that broke a record in the same state and on the same side of the tape at respective times t_1 and t_2 with $t_1 < t_2$. Let p_1 and p_2 be the tape positions of these records. Let L be the distance between r_1 and r_2 (Definition 2.5). If the content of the tape in r_1 at distance L of p_1 is the same as the content of the tape in r_2 at distance L of p_2 then $\mathcal{M}$ never halts.

Proof. Let's suppose that the record-breaking configurations are on the right-hand side of the tape. By the hypotheses, we know the machine is in the same state in r_1 and r_2 and that the content of the tape at distance L to the left of p_1 in r_1 is the same as the content of the tape at distance L to the left of p_2 in r_2. Note that the content of the tape to the right of p_1 and p_2 is the same: all-0 since they are record positions. Furthermore, by Definition 2.5, we know that distance L is the maximum distance that $\mathcal{M}$ can travel to the left of p_1 between times t_1 and t_2. Hence that after r_2, since it will read the same tape content the machine will reproduce the same behavior as it did after r_1 but translated at position p_2: after $t_2 - t_1$ steps, there will be a record-breaking configuration r_3 such that the distance between record-breaking configurations r_2 and r_3 is also L (Definition 2.5). Hence the machine will keep breaking records to the right forever and will not halt. Analogous proof for records that are broken to the left. □

Theorem 2.7. Let $\mathcal{M}$ be a Turing machine and t a time limit. The conditions of Lemma 2.6 are met before time t if and only if DECIDER-TRANSLATED-CYCLERS($\mathcal{M}$,t) outputs `true` (Algorithm 2).

Proof. The algorithm consists of a main function DECIDER-TRANSLATED-CYCLERS (Algorithm 2) and two auxiliary functions COMPUTE-DISTANCE-L and AUX-CHECK-RECORDS (Algorithm 3).

The main loop of DECIDER-TRANSLATED-CYCLERS (Algorithm 2 l.18) simulates the machine with the particularity that (a) it keeps track of the last time it visited each memory cell (l.20) and (b) it keeps track of all record-breaking configurations that are met (l.21) before reaching time limit t. When a record-breaking configuration is found, it is compared to all the previous record-breaking configurations on the same side in seek of the conditions of Lemma 2.6. This is done by auxiliary routine AUX-CHECK-RECORDS (Algorithm 3).

Auxiliary routine AUX-CHECK-RECORDS (Algorithm 3, l.14) loops over all older record-breaking configurations on the same side as the current one (l.15), and only examines older configurations that are in the same state as the current one (l.16). It computes distance L (Definition 2.5) between the older and the current record-breaking configuration (l.18). This computation is done by auxiliary routine COMPUTE-DISTANCE-L.

Auxiliary routine COMPUTE-DISTANCE-L (Algorithm 3, l.1) uses the "pebbles" that were left on the tape to give the last time a memory cell was seen (field `lastTimeVisited`) in order to compute the farthest position from the old record position that was visited before meeting the new record position (l.10). Note that we discard intermediate positions that beat the old record position (l.7-8) as we know that the part of the tape after the record position in the old record-breaking configuration is all-0, same as the part of the tape after current record position in the current record-breaking position (part of the tape to the right of the red-circled green cell in Figure 2).

Thanks to the computation of COMPUTE-DISTANCE-L the routine AUX-CHECK-RECORDS is able to check whether the tape content at distance L of the record-breaking position in both record-holding configurations is the same or not (Algorithm 3, l.23). The routine returns `true` if they are the same and the function DECIDER-TRANSLATED-CYCLERS will return `true` as well in cascade (Algorithm 2 l.24). That scenario is reached if and only if the algorithm has found two record-breaking configurations on the same side that satisfy the conditions of Lemma 2.6, which is what we wanted. □

Corollary 2.8. Let $\mathcal{M}$ be a Turing machine and $t \in \mathbb{N}$ a time limit. If DECIDER-TRANSLATED-CYCLERS($\mathcal{M}$,t) returns `true` then the behavior of $\mathcal{M}$ from all-0 tape has been decided: $\mathcal{M}$ does not halt.

Proof. Immediate by combining Lemma 2.6 and Theorem 2.7. □

2.2.3 Results

The decider was coded in `golang` and is accessible at this link: `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-translated-cyclers`.

The decider found 73,860,604 "Translated cyclers", out of 88,664,064 machines in the seed database of the Busy Beaver Challenge (c.f. `https://bbchallenge.org/method#seed-database`). Time limit was set to 1000 in a first run then increased to 10000 for the remaining machines and an additional memory limit (max number of visited cells) was set to 500 then 5000. More information about these results are available at: `https://discuss.bbchallenge.org/t/decider-translated-cyclers/34`.

2.3 n-gram Closed Position Set (CPS)

2.4 Repeated Word List (RepWL)

2.5 $S(4) = 107$ and $S(2, 4) = 3,932,964$

3 The road to S(5)

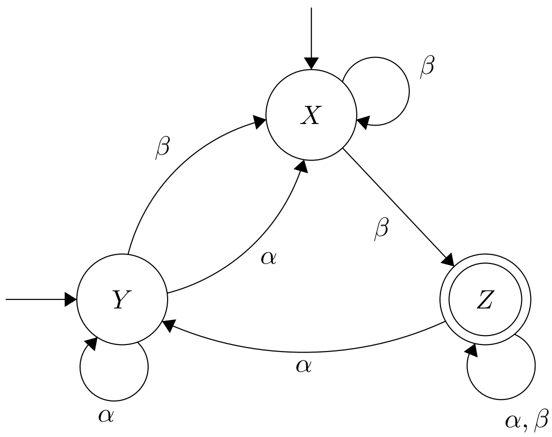


Figure 4: Example Nondeterministic Finite Automaton (NFA) with 3 states X, Y and Z, alphabet $\mathcal{A} = \{\alpha, \beta\}$, initial states X and Y, and accepting state Z. The linear-algebra representation of this NFA is given in Example 3.1. Example accepted words are: β, $\alpha\beta$, $\alpha\alpha\beta\beta$. Example rejected words are: α, $\alpha\alpha$, $\alpha\alpha\alpha$.

3.1 Finite automata reduction (FAR)

Acknowledgement. Sincere thanks to bbchallenge's contributor Justin Blanchard who initially presented this method and the first implementation[2]. Others have contributed to this method by producing alternative implementations (see Section 3.1.5) or discussing and writing the formal proof presented here: Tony Guilfoyle, Tristan Stérin (cosmo), Nathan Fenner, Mateusz Naściszewski (Mateon1), Konrad Deka, Iijil, Shawn Ligocki.

3.1.1 Method overview

The core idea of the method presented in this section is to find, for a given Turing machine, a regular language that contains the set of the machine's eventually-halting configurations (with finitely many 1s). Then, provided that the all-0 configuration is not in the regular language, we know that the machine does not halt.

A dual idea has been explored by other authors under the name Closed Tape Languages (CTL) as described in S. Ligocki's blog [3] and credited to H. Marxen in collaboration with J. Buntrock. The CTL technique for proving a Turing machine doesn't halt is to exhibit a set C of configurations such that:

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

If such a set C exists then the machine does not halt. The CTL approach has proven to be practical and powerful when we search for C among regular languages [3] [2].

Here, we develop an original *co-CTL* technique[4], based on the algebraic description of Nondeterministic Finite Automata (NFA), for finding a regular language which contains a machine's eventually halting configurations (in general a superset).

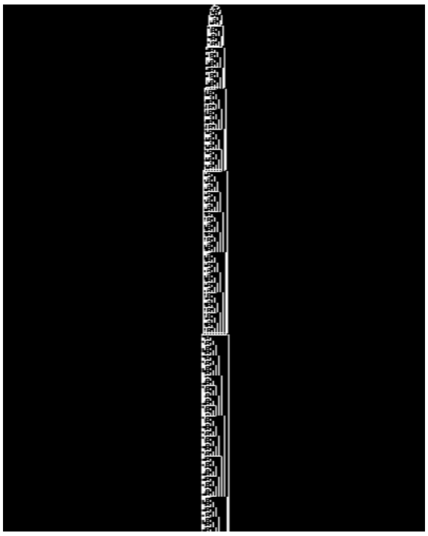
One important aspect of the technique is that, given a Turing machine and its constructed NFA—if found—it is a computationally simple task to verify that the NFA's language does indeed recognise all eventually-halting configurations (with finitely many 1s) of the machine.

[2]See: `https://discuss.bbchallenge.org/t/decider-finite-automata-reduction/`.

[3]Criteria 1–2 give a strict definition; in [3], C only needs to contain some descendant of the initial configuration and some descendant of the successor to each $c \in C$. In that case, the set of ancestor configurations to those in C meets the strict definition.

[4]By co-CTL we mean a set whose complement is a CTL, characterized by closure criteria inverse—or equivalently converse—to 1–3. In other words, a co-CTL contains all halting configurations, any configuration which can *precede* any member configuration by one TM transition, and not the initial configuration.

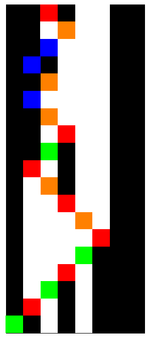
[5]`https://bbchallenge.org/1RB0LD_1LC1RA_0RB0LC_---1LA`, the machine exhibits a non-trivial counting behavior.



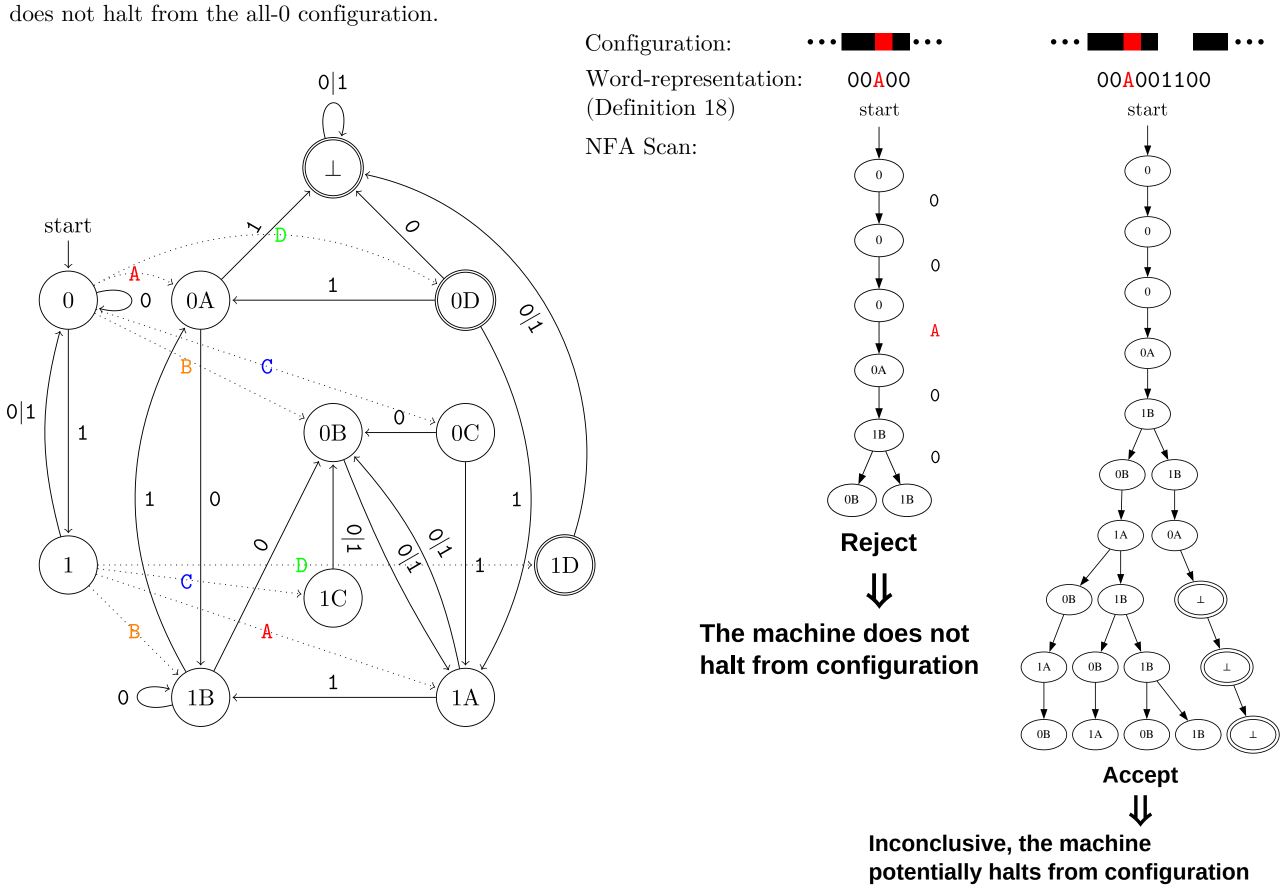
(a) 10,000-step space-time diagram of the 4-state Turing machine given in (b) from the all-0 initial configuration. The machine does not halt from the all-0 configuration.

	0	1
A	1RB	0LD
B	1LC	1RA
C	0RB	0LC
D	- - -	1LA

(b) Transition table.



(c) Detailed space-time diagram of the Turing machine given in (b) from an eventually-halting configuration: the machine halts after 18 steps by reading a 0 in state D.



(d) Left: Nondeterministic Finite Automaton for the Turing machine given in (b), constructed using FAR direct algorithm, see Section 6.3. By construction, if this NFA rejects a configuration, then we know that the configuration does not eventually halt, see Theorem 20. Right: The NFA rejects (i.e. no NFA accepting state is reached) the all-0 configuration, the machine does not halt from it. The NFA accepts (i.e. at least one NFA accepting state is reached) the starting (or any) configuration shown in (c) hence we cannot conclude that it is non-halting, which is consistent since it eventually halts.

Figure 5: A Nondeterministic Finite Automaton, used as follows to decide a 4-state Turing machine[5]: (a) Space-time diagram showing the first few descendants of the all-0 configuration for the machine. The machine actually runs forever from the all-0 configuration, adopting a "counting" behavior. (b) Transition table for the TM. (c) The TM halts in 18 steps from a different configuration; these 18 rows depict *eventually-halting* configurations. (d) A Nondeterministic Finite Automaton, constructed using the direct FAR algorithm (Section 3.1.3), that recognises at least all eventually-halting configurations (with finitely many 1s) of the machine. Inputting the top row of (c), encoded as word `00A001100` (see Definition 3.2), the NFA transitions by reading each successive symbol of the input, through NFA states: 0, 0, 0A, 1B, {0B, 1B}, {1A, 0A}, {0B, 1B, ⊥}, {1A, 0B, 1B, ⊥} and finally {0B, 1A, 0B, 1B, ⊥}. Since NFA state ⊥ is accepting (doubly circled in (d)), the NFA accepts `00A001100`, classifying this configuration as potentially eventually-halting. However, the NFA does not accept input `A0`, which corresponds to the all-0 configuration, hence this TM cannot halt from there.

3.1.2 Potential-halt-recognizing automata

For a given Turing machine, we aim at building an NFA that recognises at least all its eventually-halting configurations (with finitely many 1s). In other words, the NFA recognises configurations that *potentially* eventually halt, which is why we call the NFA *potential-halt-recognizing.* Importantly, if the NFA does not recognise the all-0 initial configuration then we know that the Turing machine does not halt from it. Figure 5 gives a potential-halt-recognizing NFA for a 4-state Turing machine, constructed using the results of Section 3.1.3.

Let's first recall how Nondeterministic Finite Automata (**NFA**) can be described using linear algebra. Let **2** denote the Boolean semiring[6] $\{0,1\}$ with operations $+$ and $\cdot$ respectively implemented by OR and AND [1]. Let $\mathcal{M}_{m,n}$ be the set of matrices with m rows and n columns over **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 $q_0 \in \mathcal{M}_{1,n}$ and $a \in \mathcal{M}_{1,n}$ 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 $T_\gamma \in \mathcal{M}_{n,n}$ 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 $a^T \in \mathcal{M}_{n,1}$ the transposition of a.

Example 3.1. The NFA depicted in Figure 4, with states X, Y, Z and alphabet $\mathcal{A} = \{\alpha, \beta\}$, is algebraically encoded as follows: $q_0 = (1,1,0)$, $a = (0,0,1)$, $T_\alpha = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$ and $T_\beta = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. The reader can check that words β, $\alpha\beta$ and $\alpha\alpha\beta\beta$ are accepted, i.e. $q_0 T_\beta a^T = 1$, $q_0 T_\alpha T_\beta a^T = 1$ and $q_0 T_\alpha T_\alpha T_\beta T_\beta a^T = 1$. But, words α, $\alpha\alpha$ and $\alpha\alpha\alpha$ are rejected, i.e. $q_0 T_\alpha a^T = 0$, $q_0 T_\alpha T_\alpha a^T = 0$ and $q_0 T_\alpha T_\alpha T_\alpha a^T = 0$.

Now, we describe how we transform Turing machine configurations that have finitely many 1s into finite words that will be read by our NFA. First recall that a Turing machine configuration is defined by the 3-tuple: (i) state in which the machine is (ii) position of the head (iii) content of the memory tape, see Section **??**. Then, a word-representation of a configuration is defined by:

Definition 3.2 (Word-representations of a configuration)**.** Let c be a Turing machine configuration with finite support, i.e. there are finitely many 1s on the memory tape of the configuration. A word-representation of the configuration c is a word $\hat{c}$ constructed by concatenating (from left to right) the symbols of any finite region of the tape that contains all the 1s, and adding the state (a letter between A and E in the case of 5-state TMs) just before the position of the head.

Example 3.3. A word-representation of the configuration on the top row of Figure 5(c), is $\hat{c} =$ `00A001100`.

Note that two word-representations of the same configuration will only differ in the number of leading and trailing 0s that they have. Hence, if $\mathcal{L}$ is the regular language of the NFA that we wish to construct to recognise the eventually-halting configurations (with finitely many 1s) of a given TM, it is natural that we require the following:

$$u \in \mathcal{L} \iff 0u \in \mathcal{L} \qquad \text{(leading zeros ignored)}$$
$$u \in \mathcal{L} \iff u0 \in \mathcal{L} \qquad \text{(trailing zeros ignored)}$$

These are implied by the following, generally stronger, conditions on the transition matrix $T_0 \in \mathcal{M}_{n,n}$:

$$q_0 T_0 = q_0 \tag{3.1}$$
$$T_0 a^T = a^T \tag{3.2}$$

Note that Condition 3.2, $T_0 a^T = a^T$, means that for all accepting states of the NFA, reading a 0 is possible and leads to an accepting state. Indeed, $T_0 a^T$ describes the set of NFA states that reach the set of accepting states a after reading a 0.

[6] A semiring is a ring without the requirement to have additive inverses, e.g. the set of natural numbers $\mathbb{N} = \{0,1,2\dots\}$ is a semiring.

Then, we want our NFA's language $\mathcal{L}$ to include all eventually-halting configurations (with finitely many 1s) of a given Turing machine $\mathcal{M}$. Inductively, we want that:

$$c \vdash \bot \implies \hat{c} \in \mathcal{L}$$
$$(c_1 \vdash c_2) \wedge \hat{c}_2 \in \mathcal{L} \implies \hat{c}_1 \in \mathcal{L}$$

With c, c_1, c_2 configurations of the TM (with finite support) and $\hat{c}, \hat{c}_1, \hat{c}_2$ any of their finite word-representations, see Definition 3.2. Let $f, t \in \{A, B, C, D, E\}$ denote TM states (the "from" and "to" states in a TM transition), and $r, w, b \in \{0, 1\}$ denote bits (a bit "read", a bit "written", and just a bit), then the above conditions turn into:

$$\forall u, z \in \{0,1\}^* : \; ufrz \in \mathcal{L}, \text{ if } (f, r) \to \bot \text{ is a halting transition of } \mathcal{M}$$
$$\forall u, z \in \{0,1\}^*, \forall b \in \{0,1\} : utbwz \in \mathcal{L} \implies ubfrz \in \mathcal{L}, \text{ if } (f, r) \to (t, w, \text{left}) \text{ is a transition of } \mathcal{M}$$
$$\forall u, z \in \{0,1\}^*, \forall b \in \{0,1\} : uwtz \in \mathcal{L} \implies ufrz \in \mathcal{L}, \text{ if } (f, r) \to (t, w, \text{right}) \text{ is a transition of } \mathcal{M}$$

Which algebraically becomes:

$$\forall u, z \in \{0,1\}^* : \; q_0 T_u T_f T_r T_z a^T = 1, \text{ if } (f, r) \to \bot \text{ is a halting transition of } \mathcal{M}$$
$$\forall u, z \in \{0,1\}^*, \forall b \in \{0,1\} : q_0 T_u T_t T_b T_w T_z a^T = 1 \implies q_0 T_u T_b T_f T_r T_z a^T = 1, \text{ if } (f, r) \to (t, w, \text{left}) \text{ is a transition of } \mathcal{M}$$
$$\forall u, z \in \{0,1\}^*, \forall b \in \{0,1\} : q_0 T_u T_w T_t T_z a^T = 1 \implies q_0 T_u T_f T_r T_z a^T = 1, \text{ if } (f, r) \to (t, w, \text{right}) \text{ is a transition of } \mathcal{M}$$

These conditions are unwieldy. Let's 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.
- 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: $\forall q \in \mathcal{M}_{1,n} \; q \succeq s \implies \forall z \in \{0,1\}^* : qT_z a^T = 1$. 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 $(f, r) \to \bot$ a halting transition.

Combining the above, we get our main result:

Theorem 3.4. Machine $\mathcal{M}$ doesn't halt from the initial all-0 configuration if there is an NFA $(q_0, \{T_\gamma\}, a)$ and row vector s satisfying the below:

$q_0 T_0 = q_0$	(leading zeros ignored)	(3.1)
$T_0 a^T = a^T$	(trailing zeros ignored)	(3.2)
$sa^T = 1$	(s is accepted)	(3.3)
$sT_0, sT_1 \succeq s$	(s is a steady state)	(3.4)
$\forall u \in \{0,1\}^* : q_0 T_u T_f T_r \succeq s$	if $(f, r) \to \bot$ is a halting transition of $\mathcal{M}$	(3.5)
$\forall b \in \{0,1\} : T_b T_f T_r \succeq T_t T_b T_w$	if $(f, r) \to (t, w, \text{left})$ is a transition of $\mathcal{M}$	(3.6)
$T_f T_r \succeq T_w T_t$	if $(f, r) \to (t, w, \text{right})$ is a transition of $\mathcal{M}$	(3.7)
$q_0 T_A a^T = 0$	(initial configuration rejected)	(3.8)

Proof. Conditions (3.1)–(3.7) ensure that the NFA's language includes at least all eventually halting configurations of $\mathcal{M}$. Condition (3.8) ensures that the initial all-0 configuration of the machine is rejected, hence not eventually halting. Hence, if conditions (3.1)–(3.8) are satisfied, we can conclude that $\mathcal{M}$ does not halt from the initial all-0 configuration. □

Remark 3.5 (Verification)**.** Theorem 3.4 has the nice property of being suited for the purpose of *verification*: given a TM, an NFA and a vector s, the task of verifying that equations (3.1)–(3.8) hold and thus that the TM does not halt, is computationally simple[7]. Verifiers have been implemented for Theorem 3.4, see Section 3.1.5.

[7]Note that although equation (3.5) has a $\forall$ quantifier, 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 (3.5) replacing $q_0 T_u$ per such state.

Now, we want to design an efficient search algorithm that will, for a given TM, try to find an NFA satisfying Theorem 3.4. For that search to be feasible, we impose more structure on the NFA so that (a) the search space of NFAs is smaller (b) a subset of Conditions (3.1)–(3.7) is automatically satisfied by these NFAs.

3.1.3 Search algorithm: direct FAR algorithm

We design an efficient search algorithm for Theorem 3.4 that we call the *direct FAR algorithm*. We start by adding more structure to our NFAs as follows:

1. The NFA is constructed from two sub-NFAs: one NFA responsible for handling the left-hand side of the tape (i.e. before reading the tape-head state) and one NFA for handling the right-hand side of the tape (i.e. after reading the tape-head state).
2. The sub-NFA for the left-hand side of the tape is a Deterministic Finite Automaton (DFA).
3. Edges labelled by a tape-head state are only those that start in the left-hand side DFA and end in the right-hand side NFA. Furthermore, we require that no such two edges reach the same state in the right-hand side NFA. Hence, the right-hand side NFA has at least $5l$ states with l the number of states in the left-hand side DFA.
4. In fact, we require that the right-hand side NFA has exactly $5l + 1$ states with the extra state $\perp$ that we call the *halt state*.

Example 3.6. The structure described above is followed by the NFA depicted in Figure 5(d) Left. Note that, following above Point 3, it is natural to name states in the right-hand side NFA by prepending left-hand side DFA states to the transitions' TM state letter, e.g. state 1C in Figure 5 is reached from DFA state 1 after reading TM state letter C.

This structure might seem arbitrary but it has a very nice property that we demonstrate here: once the left-hand side DFA is chosen, there is at most one right-hand side NFA (minimal for $\succeq$) such that the overall NFA satisfies Theorem 3.4.

Indeed, let's rewrite the above structural points algebraically:

1. We write the state space of the NFA as the direct sum $\mathbf{2}^l \oplus \mathbf{2}^d$ with l the number of states of the left-hand side DFA and $d = 5l + 1$ the number of states of the right-hand side NFA. Initial state is $\begin{bmatrix} q_0 & 0 \end{bmatrix}$ with $q_0 \in \mathbf{2}^l$, transitions $T_b = \begin{bmatrix} L_b & 0 \\ 0 & R_b \end{bmatrix}$ ($b \in \{0, 1\}$) with $L_b \in \mathcal{M}_{l,l}$, $R_b \in \mathcal{M}_{d,d}$ and $T_f = \begin{bmatrix} 0 & M_f \\ 0 & 0 \end{bmatrix}$ ($f \in \{A, \ldots, E\}$) with $M_f \in \mathcal{M}_{l,d}$, and acceptance $\begin{bmatrix} 0 & a \end{bmatrix}$ with $a \in \mathcal{M}_{1,d}$.
2. $(q_0, \{L_0, L_1\})$ comes from a DFA with transition function $\delta : [l] \times \{0, 1\} \to [l]$ (with $[l]$ the set $\{0, \ldots, l - 1\}$) that ignores leading zeros, i.e. $\delta(0, 0) = 0$. That ensures (3.1) of Theorem 3.4.
3. Row vectors of matrices M_f (with $f \in \{A, \ldots, E\}$) are the standard basis row vectors $e_0, \ldots, e_{5l-1} \in \mathcal{M}_{1,d}$ where basis vector e_i has its i^{th} entry set to 1 and the other entries set to 0.
4. The right-hand side NFA has *halt state* $\perp$ and $e_{5l} = e_\perp$ as its corresponding basis row vector.

For a given Turing machine, our direct FAR algorithm will enumerate left-hand side DFAs and for each, find an associated right-hand side NFA by solving Theorem 3.4 (3.1)–(3.7) for R_0, R_1, and a. If Condition (3.8) is also satisfied then, by Theorem 3.4, the Turing machine is proven non-halting and we stop the search.

For a given left-hand side DFA with transition function δ, the right-hand side NFA is constructed by rewriting Theorem 3.4 conditions (3.4)–(3.7) in the following way, where we set the accepted steady state-set to $s = \begin{bmatrix} 0 & e_\perp \end{bmatrix}$. The algebra is helped by the general observation that for any i, the condition $\text{row}_i(M) \succeq v$ with $\text{row}_i(M)$ the i^{th} row of matrix M and v some row vector, is equivalent to $M \succeq e_i^T v$ with e_i the i^{th} standard basis vector[8].

[8]This is why we asked that row vectors of matrices M_f are standard basis vectors, Point 3 above.

$$R_r \succeq (e_\perp)^T e_\perp \qquad \text{for } r \in \{0,1\} \tag{3.4'}$$

$$\forall i \in [l] : R_r \succeq \mathrm{row}_i(M_f)^T e_\perp \qquad \text{if } (f,r) \to \perp \text{ is a halting transition of } \mathcal{M} \tag{3.5'}$$

$$\forall b \in \{0,1\}, \forall i \in [l] : R_r \succeq \mathrm{row}_{\delta(i,b)}(M_f)^T \mathrm{row}_i(M_t) R_b R_w \qquad \text{if } (f,r) \to (t, w, \mathrm{left}) \text{ is a transition of } \mathcal{M} \tag{3.6'}$$

$$\forall i \in [l] : R_r \succeq \mathrm{row}_i(M_f)^T \mathrm{row}_{\delta(i,w)}(M_t) \qquad \text{if } (f,r) \to (t, w, \mathrm{right}) \text{ is a transition of } \mathcal{M} \tag{3.7'}$$

Lemma 3.7. There's a unique minimal solution (w.r.t $\preceq$) to the system of inequalities (3.4')–(3.7') and an effective way to compute it: initialize R_0, R_1 to zero, then set entries to 1 as (3.4'), (3.5') and (3.7') demand then iterate (3.6') until R_0 and R_1 stop changing.

Proof. First notice that (3.4'), (3.5') and (3.7') have their right-hand side constant (with respect to R) hence they only amount to constant lower bounds for matrices R_0 and R_1. Then note that, given any lower bound $B_0 \preceq R_0$ and $B_1 \preceq R_1$ for true solutions of the system, we have $\mathrm{row}_{\delta(i,b)}(M_f)^T \mathrm{row}_i(M_t) R_b R_w \succeq \mathrm{row}_{\delta(i,b)}(M_f)^T \mathrm{row}_i(M_t) B_b B_w$ by compatibility of $\succeq$ with the performed operations. Hence, iterating (3.6') produces an increasing, eventually stationary, sequence of lower bounds for R_0 and R_1 whose fixed point is solution to the system. ☐

Now that we have found R_0 and R_1 we need to find the set of accepting states $\begin{bmatrix} 0 & a \end{bmatrix}$ with $a \in \mathcal{M}_{1,d}$. Conditions (3.2), (3.3) of Theorem 3.4 translate to:

$$R_0 a^T = a^T \tag{3.2'}$$

$$a \succeq e_\perp \tag{3.3'}$$

Similarly, there is a unique minimal solution (w.r.t $\preceq$) to this system which is found by initially setting $a_0 = e_\perp$ then iterating $a_{k+1} = (R_0 a_k^T)^T$ until a fixed point is reached which gives the value of a. Indeed, from (3.4'), we see that the sequence $e_\perp^T \preceq R_0 e_\perp^T \preceq R_0^2 e_\perp^T \preceq \dots$ is increasing hence it reaches a fixed point, which satisfies (3.2') and (3.3').

The last condition from Theorem 3.4 that we need to satisfy is (3.8) (rejection of the initial configuration), which translates to:

$$\mathrm{row}_0(M_A) a^T = 0 \tag{3.8'}$$

By minimality, a solution of (3.2') and (3.3') will satisfy (3.8') if and only if the minimal solution exhibited above does. Hence, we check (3.8') for the minimal a that we constructed and there are two cases:

- If a satisfies (3.8') then we have found an NFA satisfying Theorem 3.4 and we can conclude that the Turing machine does not halt from the all-0 initial configuration.
- If a does not satisfy (3.8') then we cannot conclude and we continue our search for an appropriate left-hand side DFA.

This method relies on a way to enumerate DFAs. In Section 3.1.4 we give an efficient SEARCH-DFA algorithm for enumerating canonically-represented DFAs. The search space of DFAs is a tree of partial transition functions and we can skip traversing some sub-trees based on a crucial observation. Solutions R_0, R_1 and a (given by Lemma 3.7) for partial DFA transition function δ are lower bounds of solutions for any δ' that extends δ. This observation gives that if a, constructed from δ, violates (3.8') then, any a', constructed from δ' extending δ, will violate it too. Hence, in that case, descendants of δ in the DFA search tree can be skipped. This efficient pruning technique completes the method, shown below as Algorithm 4.

3.1.4 Efficient enumeration of Deterministic Finite Automata

The direct FAR algorithm (Section 3.1.3 and Algorithm 4) relies on a procedure to enumerate Deterministic Finite Automata (DFA). We first recall the formal definition of DFAs then give an efficient algorithm

Algorithm 4 DECIDER-FINITE-AUTOMATA-REDUCTION-DIRECT

```
procedure bool DECIDER-FINITE-AUTOMATA-DIRECT(TM machine, int n, bool left_to_right)
    if not left_to_right then switch all left-going transitions of the TM to right-going and vice versa
    Matrix<bool, 5*n+1, 5*n+1> R[2*n+1][2] = [[0,0],...,[0,0]]
    ColVector<bool, 5*n+1> aT[2*n+1] = 0 // aT for transpose as a is row vector in Section 3.1.3
    ▷ Basis vector indexing: for row_i(M_f) use index 5*i+f, and for e_⊥, use index 5*n.
    Initialize R[0] using (3.4') and (3.5')
    Initialize aT[0] = e_⊥^T
    procedure CheckResult CHECK(List<int> L)
        k := L.length
        R[k], aT[k] = R[k-1], aT[k-1]
        Increase R[k] using (3.7'), with (i, w) = divmod(k-1, 2)
        repeat
            Increase R[k] using (3.6'), restricted to 2*i+b < k
        until R[k] stops changing
        repeat
            aT[k] = R[k][0] · aT[k]
        until aT[k] stops changing
        if row_0(M_A) · aT[k] ≠ 0 then return SKIP
        else if k == 2*n then return STOP
        else return MORE
    return SEARCH-DFA(check)
```

(Algorithm 5) to enumerate them and to prune the search space early based on using Lemma 3.7 on partial DFA transition functions.

Textbooks define *deterministic* finite automata (on the binary alphabet, with acceptance unspecified) as tuples (Q, δ, q_0) of: a finite set Q (states), a $q_0 \in Q$ (initial state), and $\delta : Q \times \{0,1\} \to Q$ (transition function). Though NFAs generalize DFAs, they can be emulated by (exponentially larger) power-set DFAs. [4]

To put this definition in the linear-algebraic framework: identify $q_0 \in Q$ with $0 \in [n] := \{0, \ldots, n-1\}$; represent states q with elementary row vectors e_q; define transition matrices T_b via $e_q T_b = e_{\delta(q,b)}$.

As we did for transition matrices, extend δ to words: $\delta(q, \epsilon) = q$, $\delta(q, ub) = \delta(\delta(q,u), b)$.

Given a DFA on $[n]$, call its *transition table* the list $(\delta(0,0), \delta(0,1), \ldots, \delta(n-1,0), \delta(n-1,1))$.

Call $\{\delta(q_0, u) : u \in \{0,1\}^*\}$ the set of *reachable* states.

When building a larger recognizer, we expect no benefit from considering DFAs which just relabel others or add unreachable states. So motivated, we define a canonical form for DFAs: enumerate the reachable states via breadth-first search from q_0, producing $f : [n] =: Q_{\mathsf{cf}} \to Q$. Explicitly, $f(0) = q_0$ and $f(k)$ is the first of $\delta(f(0),0), \delta(f(0),1), \ldots, \delta(f(k-1),0), \delta(f(k-1),1)$ not in $f([k])$, valid until $f([k])$ is closed under transitions. This induces $\delta_{\mathsf{cf}}(q,b) \mapsto f^{-1}(f(q), b)$. (Warning: this definition and terminology aren't standard.)

Lemma 3.8. In a DFA with $(Q, q_0) = ([n], 0)$, the following are equivalent:

1. it's in canonical form ($Q_{\mathsf{cf}} \to Q$ is the identity) and ignores leading zeros (equation (3.1) or $\delta(0,0) = 0$);
2. its transition table includes each of $0, \ldots, n-1$, whose first appearances occur in order, and with each $0 < q < n$ appearing before the $2q$ position in the transition table;
3. the sequence $\{m_k := \max\{\delta(q,b) : 2q + b \le k\}\}_{k=0}^{2n-1}$ of cumulative maxima runs from 0 to $n-1$ in steps of 0 or 1, with $m_{2q-1} \ge q$ for $0 < q < n$.

Proof. 1 $\iff$ 2**:** We prove a partial version by induction: the DFA ignores leading zeros and $f(q) = q$ for $q \le k$, iff $0, \ldots, k$ have ordered first appearances in the transition table which precede appearances of any $q > k$ and occur before the $2k$ position in δ if $k > 0$. In case $k = 0$, the DFA ignores leading zeros iff 0 comes first in the transition table by definition. (The other conditions are vacuous.) In case the claim holds for preceding k, $f(k)$ is by definition the first number outside of $f([k]) = [k]$ in the transition table—if any—and the inductive step follows.

2 $\iff$ 3: If the first appearances of $0, \ldots, n-1$ appear in order, any value at its first index is the largest so far, so m_k takes the same values. The sequence m_k is obviously nondecreasing, so to be gap-free it can only grow in steps of 0 or 1. Conversely, if m_k runs from 0 to $n-1$ in steps of 0 or 1, each value $q \in [n]$ must appear in the table at the first index k for which $m_k = q$, and all preceding values in the transition table must be strictly less.

In case these equivalent conditions are true, that last observation shows that q appears before the $\delta(q, 0)$ position iff m_k reaches q by index $k = 2q-1$, or equivalently $m_{2q-1} \geq q$.

□

Corollary 3.9. $\{t_k\}_{k=0}^{\ell}$ ($\ell < 2n$) is a prefix of a canonical, leading-zero-ignoring, n-state DFA transition table iff $m_k := \max\{t_j\}_{j=0}^{k}$ runs from 0 to $m_\ell < n$ in steps of 0 or 1, and $m_{2q-1} \geq q$ (for all $2q-1 \leq \ell$).

Proof. If $\ell = 2n-1$, $\{m_k\}$ grows to exactly $n-1$ (since $m_{2(n-1)-1} \geq n-1$), and lemma 3.8 applies. Otherwise, we may extend the sequence with $t_{\ell+1} = \min(m_\ell + 1, n-1)$, the same conditions apply. □

So, Algorithm 5 searches such DFAs incrementally (avoiding partial DFAs already deemed unworkable).

Algorithm 5 SEARCH-DFA

```
enum CheckResult {MORE, SKIP, STOP}

procedure bool SEARCH-DFA(int n, function<List<int>, CheckResult> check)
Require: check(t) ≠ MORE if t is a complete (length-2n) table
    int k = 1, t[2 * n] = [0, . . . , 0], m[2 * n] = [0, . . . , 0]
    loop
        state = check(length-k prefix of t)
        if state == MORE then
            int q_new = m[k-1] + 1
            t[k] = (q_new < n and 2*q_new-1 == k) ? q_new : 0
        else if state == SKIP then
            repeat
                if k ≤ 1 then return false
                k -= 1
            until t[k] ≤ m[k-1] and t[k] < n-1
            t[k] += 1
        else return true
        m[k] = max(m[k-1], t[k])
        k += 1
```

3.1.5 Implementations and results

Here are the implementations of the decider that were realised:

1. Justin Blanchard's original, optimized Rust implementation: `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-finite-automata-reduction`
2. Tony Guilfoyle's C++ reproduction: `https://github.com/TonyGuil/bbchallenge/tree/main/FAR`
3. Tristan Stérin (cosmo)'s Python reproduction: `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-finite-automata-reduction-reproduction`

Verifiers for Theorem 3.4 – i.e. programs that check that a given NFA gives a valid nonhalting proof for a given machine, see Remark 3.5 – have also been given with each of the above deciders and, Nathan Fenner provided one verifier formally verified in Dafny: `https://github.com/Nathan-Fenner/busy-beaver-dafny-regex-verifier`.

Results. In order to achieve reasonable compute time, the DFA search space (see Section 3.1.3) was searched up to 6 DFA states. The method decides 503,169 machines out of the 535,801 remaining machines (94%) after halting segment, in a bit less than 30 minutes using Justin Blanchard's Rust implementation on a 4-core i7 laptop. Hence, after FAR, we have 32,632 machines left to be decided.

3.2 Meet-in-the-middle weighted FAR (MitM WFAR)

3.3 Sporadic 5-state Turing machines

3.4 $S(5) = 47,176,870$

4 Beyond S(5)

A law of Busy Beaver study is that those who prove $BB(n) = x$ claim it is impossible to do the same for $BB(n+1)$. Despite this ignominious history, we cautiously posit $BB(6)$ will be extremely hard, if not impossible, to prove. Unlike previous investigators, we base our claim not on the strength of modern computation and the galatic size of large TMs (the current $BB(6)$ champion runs for far, far more steps than there are atoms in the universe), but on the mathematical hardness of the machines that remain. We call mathematically difficult TMs *cryptids*, and will discuss them later in this section. We know of cryptids in every part of the Busy Beaver frontier $BB(2,5)$, $BB(3,3)$, and $BB(6,2) = BB(6)$. Our canonical example is "Antihydra," a machine which encodes a problem very similar to the infamous Collatz Conjecture.

Example 4.1. "Antihydra"
"Antihydra" 1RB1RA_0LC1LE_1LD1LC_1LA0LB_1LF1RE_0RA
Let $f(n) = n + \lfloor \frac{n}{2} \rfloor$, or,

4.1 $S(6)$ cryptids

4.2 $S(3,3)$ cryptids

4.3 $S(2,5)$ cryptids

4.4 The Beaver Mathematical Olympiad (BMO)

5 Conclusion: the busy beaver frontier

References

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

[2] Iijil. Bruteforce-ctl. `https://github.com/Iijil1/Bruteforce-CTL`, 2022.

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

[4] M. Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1st edition, 1996.